Simulation Methods for the Development of Modular Strategic Guidance Systems

by

ENS Stephen Michael Long, USNR

B.S. Systems Engineering
United States Naval Academy, 2001

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2003

Signature of Author _____
Department of Aeronautics and Astronautics
May 23, 2003

Certified by _____
Andrew J. Staugler
Technical Staff
The Charles Stark Draper Laboratory, Inc.
Thesis Supervisor

Certified by _____
Peter M. Kachmar
Distinguished Member of the Technical Staff
The Charles Stark Draper Laboratory, Inc.
Thesis Supervisor

Certified by _____
George T. Schmidt, Sc.D.
Lecturer, Department of Aeronautics and Astronautics
Director, Education
The Charles Stark Draper Laboratory, Inc.
Thesis Advisor

Accepted by _____
Edward M. Greitzer, Ph.D.
H.N. Slater Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

[This page intentionally left blank.]

Simulation Methods for the Development of Modular Strategic Guidance Systems

by

ENS Stephen Michael Long, USNR

Submitted to the Department of Aeronautics and Astronautics
on May 23, 2003, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

The traditional approach to simulation-based system design results in a stovepiped development process where subsystems are developed independently and integration requirements are then levied on the system architecture. For applications like the MK6 Life Extension program, where a modular system architecture is the primary design goal, this method of simulation-based design is inadequate. Sponsored by the System Engineering System Design and Analysis Group at the Draper Laboratory, this thesis proposes an alternate top-down approach to simulation-based design, where the system architecture is established first, and the system is developed in an integrated fashion. Through this approach, the modularity requirements are then levied on the subsystems according to the integrated system architecture. In this thesis, a system-level simulation concept is developed via this approach to facilitate analysis of key guidance system design issues and evaluation of integrated system requirements. The simulation concept, which utilizes existing modeling and simulation tools, is validated through the design of a candidate guidance system and its usefulness is illustrated via selected guidance system modularity studies.

Thesis Supervisor: Andrew J. Staugler
Title: Technical Staff
The Charles Stark Draper Laboratory, Inc.

Thesis Supervisor: Peter M. Kachmar
Title: Distinguished Member of the Technical Staff
The Charles Stark Draper Laboratory, Inc.

Thesis Advisor: George T. Schmidt, Sc.D.
Title: Lecturer, Department of Aeronautics and Astronautics
Director, Education
The Charles Stark Draper Laboratory, Inc.

[This page intentionally left blank.]

ACKNOWLEDGMENT

May 23, 2003

First and foremost, I would like to thank my Lord and Savior, Jesus Christ. Without Him, none of this is possible. Philippians 4:13.

To my mother and father, I would not be where I am today if not for your persistent belief in me and the goals that I can achieve. You have both always put so much into my life and my accomplishments, and I cannot thank you enough. I know a lot of times I have not appreciated it as I should, but I know that no one could ever ask for a better set of loving parents. To my "little" brother Chad, I am so proud of you. Keep pushing hard and succeeding, and some day you will be here as well ... you have such a bright future ahead of you. I love you all.

To Anna, thank you so much for being there for me ... through all of the highs and lows. These last two years have been no picnic, and I cannot express to you how much it has meant to me to have you there every step of the way. You are such a wonderful blessing in my life. You are the love of my life. I am looking forward to many more years together with you ... I'm sure it won't get much easier in the Navy, but we'll make it just the same. Thank you for loving me the way you do. I love you.

I owe a very special thanks to the Draper Laboratory for this exceptional opportunity, and to Peter Kachmar and Andrew Staugler in particular. You have both done so much for me, and I have learned a tremendous amount of invaluable knowledge from each of you. I wish you the best of luck in your careers, and thank you for giving me such a great head start on mine. God bless.

To all my friends and fellows, thanks for motivating me to work hard and helping me to slack off—all at the same time. Ted, the great '00 guy, thanks for helping me represent good'ol USNA and introducing me to Marblehead. Don't ever forget ... there are more fighter jets in the ocean than submarines in the air. Stuart, well, all I gotta say is, "Come back Ali ... Come back Ali's sister!!!" Thanks for keeping the humor in this ordeal—don't forget about me when you get to Cali. John, I never knew that one could develop a $90+{}^{\circ}$ slice. I'm glad the whole NFO thing worked out for you—it'll be safer on the course in Charleston without you (just kid-

ding). Andrew, I'll see you in Charleston. We have to live together, so I'm not going to say anything mushy here. Kim, you know you've kept me sane. You're definitely one of the coolest people I've ever met, and I'm glad you finally came to your senses and fell in love with a submariner. We're good people. Please keep in touch. Christine, you give me hope for vegetarians around the world. But really, lose the veggie buffalo wings with the fake bones ... that's just weird. Good luck to each of you, wherever life takes you.

Finally, to my church family at the First Baptist Church in Marblehead ... You have all been a second family to me out here, and I can never repay you for the love and kindness that you have shown me. I will keep each of you forever in my prayers, and hopefully the Lord will bring me back to Marblehead some day to see all of you again. God bless, and thank you so much.

There are so many others who have blessed my life along the way ... my sponsor family in Annapolis (the McGinnis's); my neighbors who watched me grow up; my family and friends who have been there all along. I am afraid that I am out of space for thanks, but I want each of you to know how you have touched my life, and that I have not forgotten you. To everyone I have missed, thank you.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Contract N00030-03-C-0014, sponsored by the U.S. Navy and the Draper Laboratory Systems Engineering System Design and Analysis Group.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

ENS Stephen M. Long, USNR

Contents

List of Figures

List of Tables

[This page intentionally left blank.]

Chapter 1

Introduction

1.1 Motivation

In 1998, a service life extension of the platform that carries the U.S. Navy's Trident II D5 SLBM (submarine-launched ballistic missile)—the *Ohio*-class ballistic missile submarines—imparted a new service life requirement on the missile and its existing MK6 guidance system [3]. Subsequent life extension analyses of the guidance system and missile electronics have determined that there are potential limitations to the fulfillment of an extended service life, including degradation of electronic and mechanical systems, obsolescence of technology, and expensive support and maintenance costs. Rather than develop an entirely new weapon system for an aging platform, the U.S. Navy has initiated a program to upgrade the MK6 to ensure affordable lifetime supportability and the capability to adapt to future missions and improvements in technology.

Strategic guidance system designs traditionally follow an instrument-specific approach; the instruments are selected first, and the rest of the system is then designed to accomodate the instruments. The objective of this design philosophy is to achieve the highest accuracy possible with the latest sensor technology available. This philosophy is largely successful, as evidenced by the United States' legacy of consistent improvement in SLBM guidance system accuracy (shown in Table 1.1). The draw-

Guidance System	Deployed	Weapon System	CEP (nautical miles)
MK1	1960	Polaris A1	2
	1962	Polaris A2	
MK2	1964	Polaris A3	0.5
MK3	1971	Poseidon C3	0.25
MK5	1979	Trident C4	0.12-0.25
MK6	1990	Trident D5	0.06

Table 1.1: U.S. SLBM Guidance System Genealogy[1]

back is that the resulting system architectures are highly complex, tightly coupled, and completely unique; with little flexibility to support future modifications and/or upgrades. They are extremely expensive to sustain over an extended service life.

In order to ensure affordable long-term reliability, maintenance, and supportability of the Trident strategic weapon system, the objective of the MK6 Life Extension (MK6LE) program is to develop an updated version of the MK6 using a modular approach. The MK6LE will maintain the same accuracy and performance standards as the MK6, but abandon the traditional instrument-specific philosophy. Instead, the MK6LE will be developed with a flexible system architecture to support the insertion of alternate instruments, guidance functions, and system components throughout its service life. This flexible guidance system design will provide for spare capacity in terms of physical space, computing power, and infrastructure to support future missions and technologies.

The MK6LE program design plan includes the use of improved design methods, such as modeling and simulation, to enhance the system development process. When used appropriately, modeling and simulation can accelerate system development through rapid design iteration, improve system quality through expanded testing in synthetic environments, and illuminate potential reductions in resource expenditure [2]. By integrating the use of models and simulations throughout the guidance system development process, the MK6LE program will be able to evaluate hundreds of system concepts, converge on an optimal modular design, and thoroughly test the system for supportability before constructing a single prototype.

[1]Source: [7]

1.2 Objectives

Modeling and simulation is indeed a powerful approach for testing and evaluating system designs. However, the advantages of simulation-based design are most often realized after a point design (conceptual or otherwise) has been established. The traditional approach to simulation-based design begins at the subsystem level and follows a bottom-up progression. Subsystem teams are tasked with developing, testing, and validating subsystem models to meet their own functional and performance requirements independent of other subsystems. The candidate subsystem designs are then passed to the system level for integration in the simulation (and/or the system). Under this approach, the burden is placed on the system architecture to accomodate the individual requirements of each independently developed subsystem, and the power of modeling and simulation to help shape the system design is frequently overlooked.

For applications like the MK6LE, where the system architecture is the principal design driver, the traditional approach does not benefit the design objective. In order to develop a truly modular integrated system architecture, a top-down approach to simulation-based design is needed. This thesis seeks to provide that capability by accomplishing two principal objectives. First, a systems engineering approach to guidance system simulation will be developed and validated. This approach will result in the capability to rapidly develop solutions to system-level design issues and facilitate constant analysis and verification of integrated system requirements. Second, modular simulation architectures will be established to influence the system design. Since the physical subsystems are often developed from models used in simulation, subsystem models integrated in this modular simulation architecture will lead directly to the development of modularity in the guidance system. These objectives will be accomplished using existing modeling and simulation tools, and will be verified through implementation of a generic guidance system design.

1.3 Thesis Overview

Chapter 2 provides the rationale for the use of modeling and simulation in weapons system development. Chapter 3 then describes the system to be implemented in simulation—the strategic guidance system. The typical guidance system modes and functions are discussed, and the specific considerations associated with guidance system design are addressed. Chapter 4 discusses the development of the simulation framework to address those design considerations, and introduces two modular simulation architectures that will influence simulation and system modularity. The results of the successfully implemented simulations are presented in Chapter 5, along with case studies of the use of the architectures to investigate alternate system configurations. Finally, the conclusions are stated and recommendations for future work are made in Chapter 6.

Chapter 2

Modeling and Simulation

2.1 Overview

This chapter provides the impetus for the use of modeling and simulation in weapons system development. Modeling and simulation is a powerful system design approach that impacts all phases of the system development process. In order to provide a clear understanding of how a system benefits from simulation-based design, the essential terminology is introduced, and the advantages of modeling and simulation throughout the entire system development process are presented.

2.2 Definitions and Terminology

The most confusing aspect of any discussion relating to modeling and simulation capabilities is the difference between a model and a simulation. Therefore, it is important to first clarify the distinction between these two items and their associated meanings. The official DMSO (Defense Modeling and Simulation Office) definitions have been supplied in each case, along with remarks on their meaning in the context of computer modeling and simulation.

2.2.1 Models

According to DoD Directive 5000.59-M, a model is "a physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process." [1] Models generally consist of a set of mathematical equations or solution algorithms with their own assumptions, limitations, and approximations, that accept a specific set of inputs and provide a corresponding set of useful outputs. For example, a model of a strategic missile body's dynamics would consist of the equations of motion for that body derived from its physical characteristics. The model might contain assumptions and approximations with regard to the distribution of internal mass and moments of inertia, or limitations such as an inability to solve for motion in more than three degrees of freedom. Appropriate inputs to the model might include the applied force, thrust, and natural forces such as gravity and aerodynamic drag. These inputs would then be processed by the model's equations and/or algorithms to provide the desired outputs—in this case, missile acceleration.

2.2.2 Simulations

Simulations are defined by DoD Directive 5000.59-M as "method[s] for implementing... model[s] over time." [1] If one were to take the model of missile dynamics described above and implement it in software to solve for its outputs as functions of its inputs over time, the result would be a simulation. In other words, a simulation is the framework that executes a model or models in the proper order, provides timing and coordination between them, and controls inputs and outputs. Thus, a model provides the mathematical equations or algorithms that describe the behavior of a system or phenomenon in computer language, while a simulation serves as the software framework that solves for that behavior over time. Figure 2-1 illustrates the concept of a model by itself and in simulation using the example given in Section 2.2.1.

Before moving on, it is important to also distinguish usage of the term "modeling and simulation" from "models and simulations." Modeling and simulation refers to the approach of using models and simulations to solve problems analytically; it

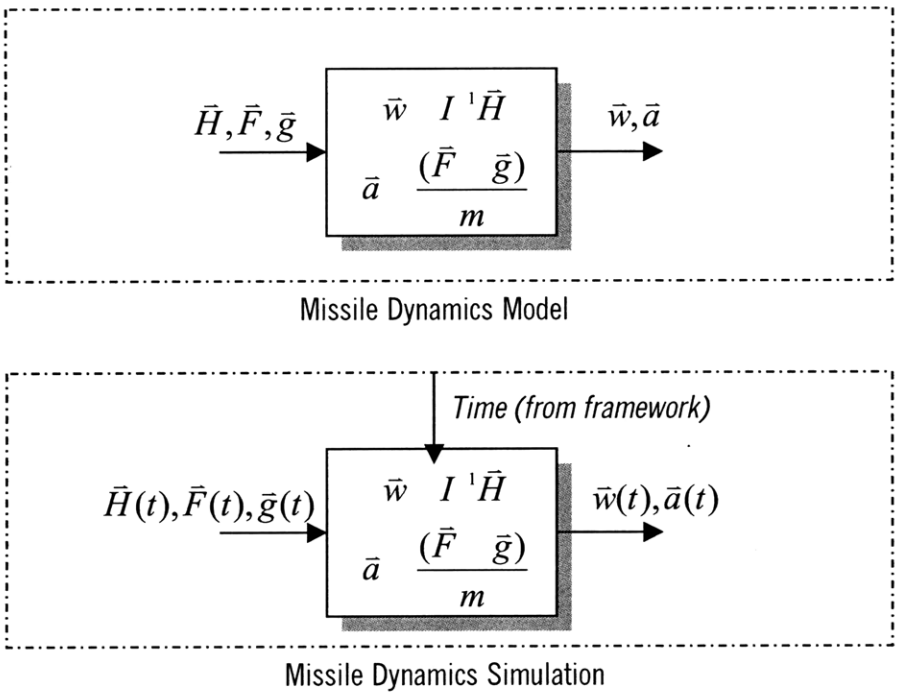

Figure 2-1: "Model" vs. "Simulation"

is a problem-solving approach. Conversely, models and simulations are the building blocks of this approach as they are defined above. These terms are often used interchangeably (particularly through use of the acronym "M&S") in modeling and simulation documentation, but they do not carry the same meaning.

2.3 Advantages of Modeling and Simulation

Modeling and simulation tools have been used to support weapon system development processes for many years. However, their use has been extremely sporadic due to limited capabilities and expensive costs. Only within the past decade have modeling and simulation tools become powerful and affordable enough to become the foundational tools of the entire system development process.

As declining budgets and shifting priorities have pressured defense acquisition programs to find better ways to develop and field new systems, the use of modeling and

simulation tools and the processes that exploit their potential benefits have expanded rapidly. The results are outlined below.

2.3.1 Improved System Quality

Through the use of modeling and simulation, the quality of the guidance system design will be improved in a number of ways. First, the capabilities of modeling and simulation tools and techniques facilitate a much more thorough investigation of the design space during concept development than previously possible. The result is the ability of the design team to evaluate hundreds of design options with respect to performance and cost and select a solution that best satisfies the requirements placed on the design. Of course, there is no guarantee that the selected design is the very best, but as the number of designs explored increases, so does the confidence that a more effective design does not exist.

System quality is also improved by the design team's ability to thoroughly test the selected design in a virtual environment. Virtual testing provides rapid feedback on the weaknesses of the system design and allows engineers to make adjustments early on in the development process. This capability combats the traditional reluctance of acquisition programs to significantly alter the system's design once a physical prototype has been constructed [5]. The relative ease of testing the system in a virtual environment also allows the design team to evaluate system performance in a number of different scenarios, thereby exposing potential pitfalls that would normally go unnoticed until deployment.

The flexibility of developing a system in a virtual environment allows last-minute modifications to be made all the way to initial production. In this manner, new systems can be continuously tweeked to address design issues as they evolve during the system's development. Finally, the use of modeling and simulation throughout the acquisition process also enables supportability considerations to be incorporated in the early stages of design, thus improving the system's ability to accomodate modifications once in service.

2.3.2 Accelerated Design Schedules

Simulation-based acquisition also increases the likelihood of developing and acquiring new systems in substantially less time than their predecessors. As stated in the previous section, the power of simulation enables iterations of design trades to occur very quickly. With this sort of rapid feedback, engineers do not have to wait for prototypes to be constructed and tested over aand simulations to develop a system in a virtual environment, thereby facilitating rapid iteration of system designs and emulation of system performance without construction of a physical prototype. substantial length of time. System assumptions can be tested and design flaws exposed in a matter of hours or days, rather than months or years.

In addition, virtual manufacturing studies allow the production community to investigate procedures that can streamline the production process, such as reducing the number of parts required. Virtual models also allow manufacturers to rehearse the manufacturing process before the system is approved for production. These practices not only lead to reduced production steps; they also minimize costly (in both time and money) manufacturing defects.

2.3.3 Reduced Acquisition Costs

A recurring advantage of modeling and simulation applications is the reduction (or perhaps even elimination) of the need for construction of physical prototypes. The ability to test and evaluate system performance in a synthetic environment reduces initial acquisition costs by diminishing the reliance on costly prototype testing programs. System models can be tested, modified, and reused again and again in simulation at no additional cost, as opposed to the inefficient "build-test-fix-test-fix" method associated with their physical counterparts [5].

Modeling and simulation capabilities can also reduce life cycle sustainment costs. It is common knowledge in the defense industry that the majority of acquisition costs are incurred through support of the system after deployment. Simulations of the system's entire life cycle provide engineers with the insight to make intelligent design

trades early on that ensure more cost-effective sustainment of the system as it ages. Similarly, three-dimensional modeling can be used by maintenance teams to assess the system's maintainability and influence the system's design accordingly, thereby producing a system that is much cheaper to operate.

2.4 Validation of Models and Simulations

The use of modeling and simulation in the system development process does not eliminate the need for construction of all prototypes. Selected prototypes are still required for new, untested technologies in order to validate the models and simulations that replicate the new technologies. Once these models and simulations are validated, the extensive power of modeling and simulation tools as discussed in Section 2.3 can be brought to bear on the system design.

Chapter 3

Strategic Guidance System Design Elements

3.1 Overview

The previous chapter provided the impetus for the use of modeling and simulation in system development. It is now important to address the fundamental elements of a strategic guidance system design, so that a sufficient simulation capability can be developed. Critical considerations in guidance system design include the system's various modes of operation, its required functionality, and considerations that support system reliability and supportability. Those considerations are addressed here in a generic fashion, in order to maximize the range of applications for the simulation approach.

3.2 Guidance System Modes

The flight of a strategic missile generally consists of two phases—a powered flight, or *boost* phase; and a reentry body deployment, or *bus* phase. Each phase can be further broken down into a series of states or *modes* during which guidance and/or missile system operation is distinct. The designation and control of these modes

by the guidance system is very important, as they are often used to sequence the weapon system through the various actions it must perform to successfully accomplish the mission. In addition, these modes can be used to develop and investigate the guidance system's operational requirements—an extremely useful technique in the simplification of guidance system design.

3.2.1 Boost Phase

Missile flight invariably begins with the burn of a series of booster stages that propel the missile into the upper atmosphere. Throughout the boost phase, the objective of the guidance system is to direct the missile's thrust such that it reaches a sufficient velocity and direction (referred to as the *correlated velocity*) to insert the reentry bodies into free-fall trajectories to their targets. Distinct guidance system operations during this phase typically correspond to the staging of the missile and any post-boost operations required in preparation for reentry body deployment (such as external position fixes ... see Section 3.3.3). Once the correlated velocity has been achieved, the last missile stage is ejected and the reentry body platform (the "bus") is in free-fall. Preparations for reentry body deployment are made, and the bus phase begins.

3.2.2 Bus (Reentry Body Deployment) Phase

During the bus phase, the guidance system assumes control of what is most often a three-axis stabilized reentry body platform. Guidance system modes frequently consist of a series of operations that are repeated for each reentry body onboard. These operations include the orientation, pointing, and release of each reentry body along a free-fall trajectory to its assigned target. Guidance calculations and commands during this phase are significantly complex, as the guidance system directs special maneuvers[1] to ensure optimal deployment conditions for each reentry body. Upon deployment of the final reentry body, the guidance mission ends.

[1] ... depending on system design.

	Mode	Mission Event
BOOST	-1	Pre-Launch Preparation
	0	Launch
	1	1st Stage Burn
	2	2nd Stage Burn
	3	3rd Stage Burn
	4	Orient for Stellar Sighting
	5	Position Update (Stellar Sighting)
BUS	6	Bus Orientation
	7	Release Reentry Body
	8	Reentry Body Avoidance Maneuver

Table 3.1: Sample Guidance System Modes

3.2.3 Mode Sequence

Table 3.1 demonstrates the manner in which a typical guidance mission might sequence through various modes during the boost and bus phases of flight. As stated earlier, these modes identify specific instances during which guidance system operation is distinct. For instance, the guidance system in this example might employ a different steering routine (see Section 3.3.6) for each stage of the missile. This explains the designation of each stage as a separate mode of the guidance system. This table also demonstrates the use of modes corresponding to separate guidance actions during reentry body deployment. For a payload of mutliple reentry bodies, the guidance system in this example would repeat Modes 6-8 until all bodies have been deployed. By considering guidance system operation in terms of modes, a straightforward approach can be taken to guidance system design and to the coordination of guidance and missile system activities.

3.3 Functional Decomposition

The guidance system must carry out a number of different activities throughout the course of a mission. These activities are typically referred to as *functions* of the guidance system. The functions presented here are divided into *primary* and

implementing functions. The primary functions are the guidance system activities carried out from a weapon system point of view, while the implementing functions are those required to perform the primary functions. Figure 3-1 reveals the system-level interaction of these functions during flight; primary functions are commonly executed sequentially (as indicated by a solid line), while implementing functions are carried out in parallel (indicated by a dashed line). As one might expect, the specific details of these functions and their execution can vary depending on system design. Therefore, only a generalized description of each of the primary functions is given here.

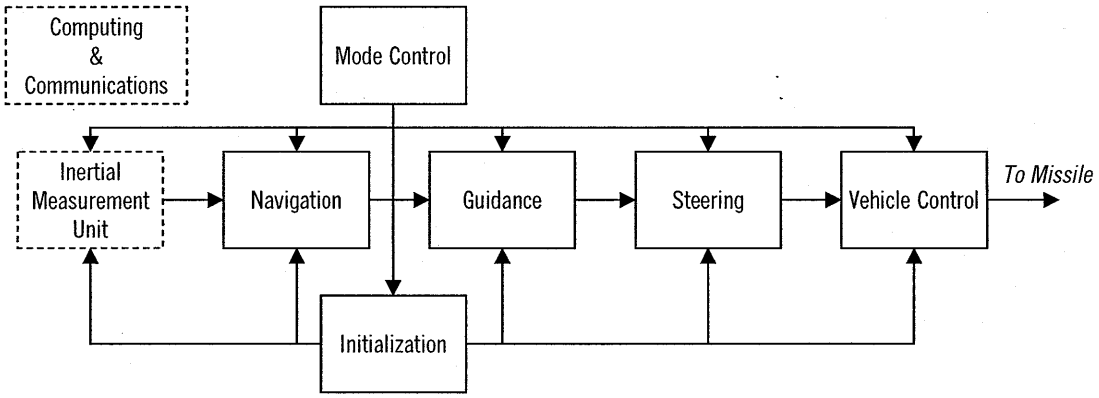

Figure 3-1: Guidance System Function Interaction

3.3.1 Mode Sequencing and Control

Section 3.2 described the use of modes to sequence the guidance and missile system through the various operations required to accomplish the mission. These modes are supervised and controlled within the guidance system by the Mode Sequencing and Control (MSC) function. The MSC function serves as the primary mission control for the entire weapon system, responsible for coordinating the operation of the various missile subsystems with guidance system activities. The role of MSC includes the direction of guidance system operations within each mode and the initiation of commands to other missile subsystems to act accordingly. In addition, the MSC function monitors the various criteria that determine the transition of the system from one

mode to another (called *transition conditions*). When transition conditions are met, MSC initiates the transition of the system to the new mode and directs any actions that must be taken to begin the new mode, such as initialization routines.

3.3.2 Initialization

Initialization routines and data loads for all guidance operations are supplied by the Initialization function. Initialization routines include the startup procedures for all guidance system components; recovery procedures to reinitialize the system after a failure (see Section 3.4.3); and mid-flight initializations, such as the configuration of functions at the beginning of a new mode. The Initialization function also supplies most program and data loads, including the target database, flight control parameters, and initial conditions for guidance calculations.

3.3.3 Attitude & Velocity Measurement

The main inputs to the guidance system's calculations are measurements of the missile's attitude and kinematic acceleration. These measurements are supplied by the Attitude & Velocity Measurement function. This function is typically implemented by a cluster of instruments—usually a combination of gyroscopes and accelerometers—that are collectively referred to as the inertial measurement unit (IMU). The IMU records inertia accelerations with respect to inertial space via the accelerometers, while the gyros measure the rotation rate of the accelerometers in an inertial frame[2]. Thus, the Attitude & Velocity Measurement function establishes a physical coordinate frame and the means to measure the missile's motion within that frame. These measurements are in turn used by the guidance computers to derive missile attitude, position, and velocity.

[2]For more information on inertial instrumentation, see [4]

3.3.4 Navigation

The Navigation function uses the measurements supplied by the inertial measurement unit to estimate the missile's position and velocity in an inertial frame. The missile's position and velocity is generally computed by solving the equations:

$$\vec{V_m} = (\vec{f} + \vec{g})dt + \vec{V_0} \tag{3.1}$$

$$\vec{R_m} = \vec{V_m}dt + \vec{R_0} \tag{3.2}$$

where $\vec{f}$ is the specific applied force as measured by the instruments over the time interval dt and $\vec{g}$ is the gravitational acceleration based on a particular mathematical gravity model. The initial conditions $\vec{R_0}$ and $\vec{V_0}$ are the launch position and velocity supplied by initialization or the estimates of the last navigation calculation.

3.3.5 Guidance

The Guidance function uses the missile position and velocity supplied by Navigation to issue steering and thrust commands to the missile's vehicle control during boost. It does so by solving for the correlated velocity first mentioned in Section 3.2 and comparing this velocity to the missile's current velocity. The correlated velocity, $\vec{V_c}$, is calculated via the solution of a Lambert Problem involving the missile's current position, the target position, and the time-of-flight remaining. Solution of the Lambert Problem yields an instantaneous velocity that will deliver the missile to the target in the specified time. For a detailed discussion of the Lambert Problem and solution techniques, see [8].

Once the correlated velocity has been computed, the guidance function calculates the velocity-to-be-gained, $\vec{V_g}$, that will be used by steering to direct the missile's thrust. The velocity-to-be-gained is calculated via the simple vector subtraction

$$\vec{V_g} = \vec{V_c} - \vec{V_m} \tag{3.3}$$

where $\vec{V}_m$ is the current missile velocity. The Guidance function may also be designed to calculate additional steering parameters such as missile roll rate and pointing directions.

As stated earlier, the Guidance function becomes considerably complex during the reentry body deployment phase. The operation of the Guidance function during this phase depends so much on system design that few generalizations can be made. It is sufficient to say that the role of the Guidance function throughout the bus phase is to calculate the correlated velocity for each reentry body, generate orientation commands to the bus, and direct deployment of the reentry body. Additional considerations are beyond the scope of this study.

3.3.6 Steering

Various considerations during missile flight often require that complex steering routines be used to ensure missile safety and enable guidance system operations. For instance, the missile's attitude rate of change during transit through the dense lower atmosphere is often restricted to prevent aerodynamic stresses from causing structural breakup. This is called minimum-impulse steering. During the final stage, generalized energy management steering (GEMS) may be used to ensure that all thrust is expended just as the correlated velocity is reached. Or, as a final example, a post-boost position update may require roll stabilization of the orbiting bus. These steering routines are all carried out by the Steering function, which generates commands to Vehicle Control based upon guidance calculations and mode.

3.3.7 Vehicle Control

The final function in the primary sequence is the Vehicle Control function. Vehicle Control interfaces with the missile/bus propulsion system to execute the actuation commands generated by Steering. Operation of the Vehicle Control function during boost includes manipulation of thrust nozzle deflections and command of stage separation and ignition pyrotechnics. During the bus phase, Vehicle Control usu-

ally provides full attitude and translation control of the reentry body platform via a system of attitude control thrusters.

3.3.8 Computing & Communications

In order to perform all of the functions listed in this section, the guidance system must possess adequate processing and memory capabilities. In addition, guidance system computations must be reliable and accurate to a degree far greater than most weapon systems, as they have no human backup. These capabilities are provided by the Computing function, which is carried out by a host of radiation-hardened processors, circuit boards, and memory arrays housed in an electronics assembly (EA). The primary guidance functions (MSC, Navigation, Guidance, etc.) are typically implemented in software and executed by the onboard digital processors. The implementing functions are characteristically implemented in hardware and interfaced with the guidance computers via analog-to-digital and digital-to-analog converters provided by the Communications function. The Communications function includes all of the infrastructure that allows the passing of data between the various guidance components as well as between the guidance system and other missile subsystems. It will become evident in Section 3.4.2 that the Computing and Communications functions play a particularly vital role in determining guidance system supportability, as they must be structured to provide flexibility to adapt to mission modifications and hardware changes over the lifetime of the system.

3.4 System-Level Design Considerations

The development of a strategic guidance system design extends far beyond the determination of operational requirements and the functions that will carry out the guidance system activities. System developers must carefully coordinate the execution of the functions described above and ensure that sufficient capacity exists for those functions to communicate with each other. To ensure system reliability, the

critical states of the guidance system's calculations must be identified and protected so that the system can recover from a massive disruption.

3.4.1 Task Scheduling

As with any complex digital and analog system, the various functions of the guidance system are executed at different rates and, most likely, on different processors throughout the system. To get all of these components and functions to work together, a scheduling system must exist that ensures each function is executed at the proper time and in the proper order. This system is commonly referred to as the *task scheduler.*

Several considerations are involved in the design of task scheduling routines. First, the execution rates for each function must be determined. These rates are generally based upon a compromise of system performance and bandwidth capacity. Supervisory functions, such as Mode Sequencing and Control, generally require execution at higher rates, while guidance calculations (Navigation, Guidance, etc.) can be updated at much lower rates. Second, the proper execution order must be determined based upon the flow of system data. In many cases, this involves sequencing of functions distributed amongst several processors. Finally, the execution time of each function must be accounted for. If a function takes too long to execute, it can disrupt the flow of system data and cause system failure. These functions must be broken up and executed in mutliple installments to avoid such complications. Figure 3-2 demonstrates the re-partitioning of a function (Function C) that runs too long.

3.4.2 Communications and Bandwidth

System developers must be able to determine the rate and size of data passed between subsystems to ensure that adequate bandwidth exists in the communications infrastructure between subsystems. If there is not enough bandwidth, the communications capacity must be expanded, or the rate or amount of data being passed must be reduced. Considerations must also be made for the provision of spare capacity to

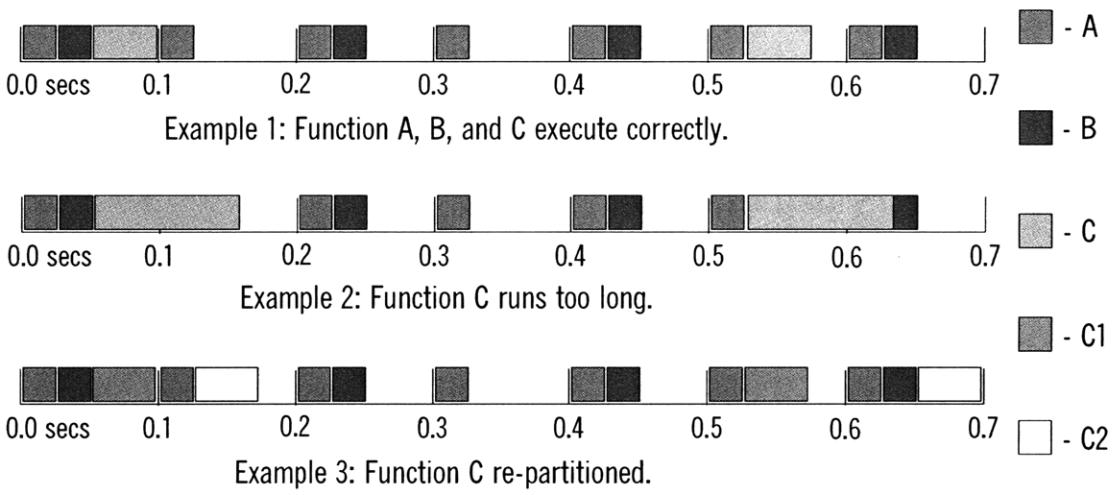

Figure 3-2: Task Scheduling Concepts

account for future hardware or software changes.

3.4.3 Circumvention and Recovery

The significance of a strategic guidance system's mission means that it must carry a high tolerance to survive massive system disruptions. Part of the establishment of this capability is the determination of the critical guidance system states that must be protected during a disruption. These states are the conditions stored by guidance system functions to carry out their activities. Stored data like elapsed time-in-flight, last position and velocity calculations, accumulated velocity measurements, etc., are all critical states that are required to update guidance calculations. In the event that the system suffers a massive disruption, these states provide the necessary initial conditions to tell the system where it left off. In addition, routines must exist to properly reinitialize the system after such a failure.

3.4.4 Modular Architecture Considerations

A modular guidance system architecture can be reasonably defined as an architecture that can accomodate alternate components, functions, and/or mission concepts with a minimal set of changes. To minimize the potential for system modifications in

these instances, there are certain desirable qualities that can be used as guidelines to drive key system design decisions. These guidelines include separation of the system by mode, reduction of subsystem interactions, and partitioning along physical and functional lines.

Recall from Section 3.2 that distinct states called modes are used to characterize guidance system operation over the course of a mission. Section 3.3.1 described how modes are useful in the coordination and control of guidance system activities with the rest of the weapon system. For these reasons, organization of the system architecture by mode is often an effective means of improving system modularity. A mode-specific architecture facilitates simplified understanding of the system. It allows system operation to be defined and studied by mode. Mode-specific architectures also simplify functional packaging, enabling system designers to separate functions by mode, rather than integrating all mode-based activities within a single function. This greatly improves the supportability of functions, because function interactions and requirements are simplified and can be modified without fear of adversely impacting other modes. Finally, the mode-specific approach means that coordination and control of the guidance system modes can be centralized at the system level, rather than distributed throughout the subsystems.

System modularity is also enhanced through the minimization of subsystem interactions. These interactions can be reduced in a number of ways. One way is to implement standardized data conventions that eliminate the need to pass additional parameters. An example of this is the use of a standard coordinate frame to pass all guidance calculations. Another method is to package functions or subsystems with complex interactions together. This generally involves physical or functional partitioning, described in the next section. By minimizing subsystem interactions, the number and extent of system modifications required to incorporate changes is reduced.

One of the most effective methods of achieving system modularity is careful partitioning of the system. The system may be partitioned physically—that is, the subsystems may be grouped physically according to location or functionality—in or-

der to accomplish modularity with respect to component replacement or maintenance. The system may be partitioned functionally. This includes the grouping or splitting of functions according to subsystem, activity, or location, in order to facilitate simplified modifications and upgrades. The fundamental effect of these forms of partitioning often boils down to the minimization of subsystem interactions, and a modular system is often the result of a mixed application of these methods.

Chapter 4

Development of the Simulation Capability

4.1 Overview

The fundamental elements of a generic strategic guidance system along with system-level design considerations were defined in the previous chapter. The purpose of this chapter is to demonstrate the application of a systems engineering approach to the development of a simulation that will (1) allow investigation of key guidance system design issues; and (2) provide an actual framework for system design. The simulation will then be implemented in a modular architecture with the intention of influencing the system development towards a more modular approach. As stated in the objectives of this thesis, the simulation is developed using existing modeling tools.

4.2 The Simulation Tool

The simulation tool used in this research is the Simulink^{TM} software package. Simulink is a model-based design tool produced by The Mathworks, Inc., and designed to work with the MATLAB^{TM} mathematical suite.

The Simulink package is an appropriate simulation tool for this application for several reasons. One of its primary features is the provision of a graphical interface for building models as block diagrams. Through this interface, complex guidance system components and missile dynamics can be constructed by clicking on and connecting prefabricated blocks, rather than formulating lengthy sets of differential equations in a compiled language. Simulink also supports the use of compiled languages. Through the use of a simple code extension, system algorithms written in compiled languages can be directly imported into the simulation from the actual system. Another critical facet of Simulink is its ability to support multirate systems—an absolute requirement for any simulation framework being used to develop a strategic guidance system. Finally, Simulink's integration with the MATLAB suite provides a vast array of efficient analysis tools that can be readily employed by the simulation at any point. Guidance system parameters can be monitored at any interface in the system, and performance data can be stored for postprocessing in MATLAB.

The drawbacks of Simulink are the same as any other computer simulation tool. First, the nature of Simulink as a graphical interface means that some modifications must be made to each model to integrate it in the simulation. Every effort must be made throughout the development of the simulation to ensure that the behavior of the system being modeled is not an artifact of the method of implementation in the simulation framework. Second, the virtual world of computer simulation is a perfect one. Thus, potential design flaws can often be hidden by the ideal operation of a system in simulation. Thorough consideration must be given to methods that expose these flaws, even in a controlled environment.

Additional features and limitations of the simulation tool will be addressed as they arise in the development of the simulation. At various points throughout this discussion, specific features of Simulink will be introduced. When those features are introduced, `a typewriter font` is used to signify that they are a unique capability of Simulink.

4.3 The Approach

The traditional systems engineering process is based on a top-down, hierarchical decomposition of system requirements. In other words, system requirements are determined first, and then used to allocate requirements at the subsystem level. The application of this process to simulation means that the desired capabilities of the simulation are established and then used to drive the development of the simulation framework. This top-down approach facilitates isolation of the simulation architecture by establishing the framework first, and then levying integration requirements on the subsystem models. The simulation architecture can then be implemented in a modular fashion to enhance the development of system modularity.

The requirements for this simulation are derived from the guidance system design considerations discussed in Section 3.4—task scheduling, communications evaluation, and circumvention and recovery. These system-level considerations not only represent critical aspects of the guidance system design that need to be investigated—they are unique requirements that define how the simulation framework must be constructed.

4.3.1 Framework Development

It is important not to confuse the simulation framework with the simulation tool. The simulation tool provides the medium for implementation and execution of the models; it supplies timing for the solution of model behavior, memory for variable storage, and the coordination of model inputs and outputs. The framework is the infrastructure implemented in the simulation by the user that tailors the simulation tool to meet the user's requirements. For example, Simulink provides timing to all models and can determine proper execution order on its own. The guidance system requires the execution of models at multiple rates and the ability to control those rates as well as the execution order. Consequently, a task scheduling framework must be developed by the user to transform the timing provided by Simulink into a timing system that replicates the operation of a guidance system. The same is true for communications identification and data storage.

The framework must also be developed in a manner that does not impact the system design, and facilitates modularity studies. The task scheduling framework should provide an easily accessible method of altering execution rates, order, and calculation time, without modification of the function itself. The communications infrastructure should facilitate minimized interfaces, and provide ready identification of necessary system modifications when required. Finally, the data handling in simulation must be treated as it is in the physical system; critical states must be stored to designated memory outside of the function, and the function must be implemented without modifications to the algorithm and/or code.

Task Scheduling

The objective for the simulation task scheduling framework is to provide the means for system designers to generate task scheduling routines for the system. That is, the simulation must enable the user to determine which functions require higher (or can accomodate lower) execution rates, which functions must be broken up to avoid overstepping interrupts, and the proper execution order of these functions. The difficulty in developing a simulation framework that achieves these objectives is that some aspects that occur rather naturally in the physical system—such as the generation of frequency interrupts or the time it takes for a function to execute—are not easily replicated in non-real time simulation. In order to develop legitimate routines that account for physical system considerations (see Section 3.4.1), the simulation requires that a mechanism be developed that can issue time interrupts, execute models discretely, control the order of execution, and simulate function runtime/communications delays.

The fundamental component of the task scheduling framework is the timing source. In the guidance system, a precision oscillator provides a frequency reference which is used to determine timing interrupts that drive the electronics and computers. A similar mechanism can be achieved in Simulink through the use of `pulse generators`. These model blocks generate signal pulses at configurable widths and rates as entered by the user, based upon the time supplied by Simulink. For each rate required by

the system in simulation, a corresponding pulse generator block is required. These blocks are grouped into a subsystem and form the single timing source (a simulated oscillator) for the entire simulated system.

With an adequate timing source established, the task scheduler can now be developed. The task scheduler will use the timing interrupts to execute the guidance system functions according to the execution rate established for each. Because the functions are represented in the simulation by models, some type of trigger signal must be used to switch the models on and off (to avoid continuous execution of the models by Simulink). Upon first glance, it might appear that the pulse generators could be used to provide the rate and the trigger signal simultaneously. However, these blocks alone are insufficient for one significant reason—their dependence on time. The pulse width is calculated as a percentage of the frequency of the pulse (the duty cycle). Thus, the signal used to trigger function execution would be on for a certain period of time, and off for the rest. If the function is to be executed only once per interrupt, the pulse width must be smaller than the incremental time step taken by Simulink's internal solvers. Otherwise, the trigger signal will be on when Simulink takes a second time step, and the function will be executed twice despite the fact a new interrupt has not been generated (see Figure 4-1).

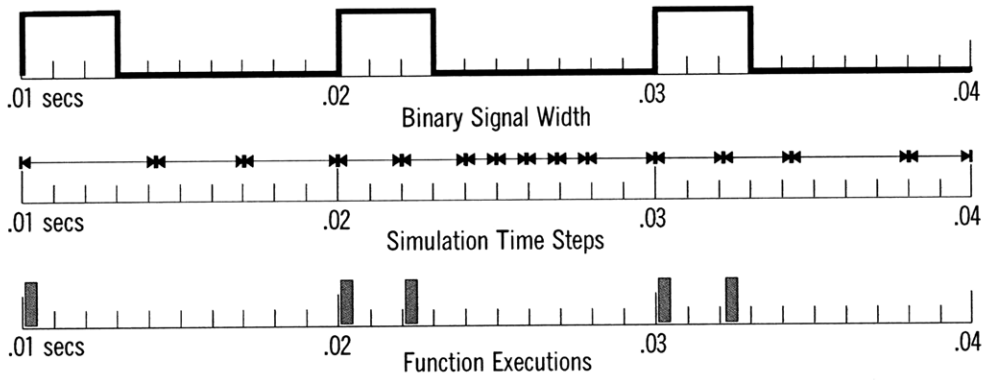

Figure 4-1: Multiple Same-Pass Executions with Pulse Signal Triggers

The time step taken by Simulink can be controlled through use of a fixed-step solver, but this type of solver has two major disadvantages. First, continuous models

(like the missile body's equations of motion) cannot be executed using fixed-step solvers; and second, the fixed-step solver greatly reduces the speed of the simulation[1]. The pulse generators also lack the means to ensure execution order or simulate a function's runtime. Therefore, this method of task scheduling is insufficient.

In order to provide the capabilities required of the task scheduler, it is already apparent that complex and supervisory control logic is required. The most efficient way to implement this sort of logic in simulation is through the use of a state chart. State charts provide a graphical representation of complex logic, and are subsequently often easier to understand and modify. State charts developed in Simulink (through use of a tool called Stateflow^{TM}) lend an added advantage because they can generate `function-call` triggers, which execute models as if they were routines called in a compiled language (only once per call).

Recall from Section 3.4.1 that an executive is a routine that establishes function rates and execution order in each mode. Figure 4-2 depicts the top-level state chart used to execute the task scheduling routine for each mode of the system. A chart like the one shown in Figure 4-3 is created for each function in the system. The function states are connected in the execution order desired to form the *executive*. Because the operation of a state chart requires knowledge of finite state machine theory (which is far beyond the scope of this thesis), Figure 4-3 on page 44 will be used to guide the discussion of how the task scheduler works.

At the beginning of each time interrupt, Simulink enters the mode executive (Figure 4-2) and progresses to the first function (state) in the sequence. The state (Figure 4-3) is entered at the upper left corner, and immediately a path must be chosen. If the current time interrupt is not the appropriate interrupt for the current function, the state is exited via the downward path, and the next function is checked. If the current interrupt does match the function's assigned interrupt, the start time of the function is recorded and a timing loop is entered. Within the timing loop, Simulink enters the "wait" state and increments the simulation time. Once the specified execution time

[1]The preferred variable-step solver takes variable time steps depending on the rate of change of system outputs. Time steps are decreased for increased system behavior, and vice versa. This method dramatically improves simulation efficiency.

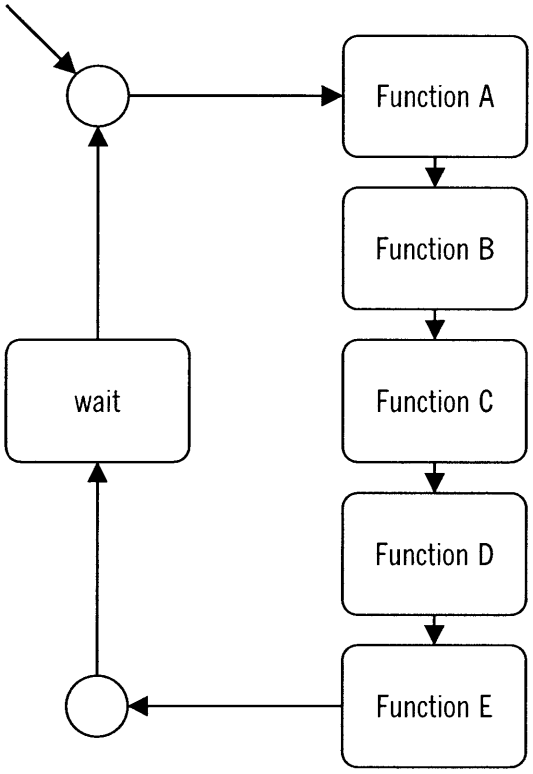

Figure 4-2: Task Scheduler State Chart

has been met, a function-call is issued to the function and its outputs are released to the other functions (via the bottom leftward path). The process then begins again with the next function in order. In this manner, the simulation is able to emulate a function's runtime and communication delay in a non-real time environment.

The described framework, although seemingly complex, provides a robust and efficient method for simulating the task scheduler. The execution order and rate of each function is readily identifiable and simple to modify. By assigning a state to each function, functions can be removed or added to the mode executive by simply deleting or inserting the appropriate state and connecting the function-call to the function model. Finally, this method allows the option of estimating runtime considerations in a non-real time environment, laying a foundation for the inclusion of communications models in the simulation.

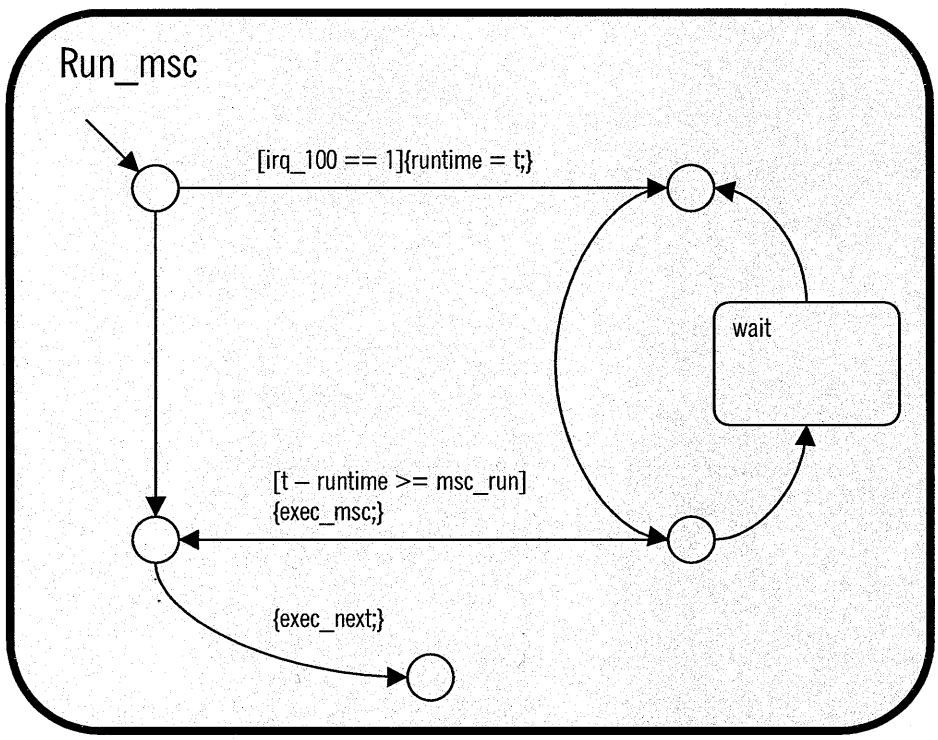

Figure 4-3: Task Scheduler - Function Execution State Chart

Communications/Interface Identification

With the framework for execution of the guidance system functions established, the next step is to develop the framework for passing data between the various subsystems. The primary objective of the communications framework is to provide the capability to readily identify the data passed between subsystems (functions or components) in simulation. This capability is a major enabler in understanding subsystem interactions and minimization of interfaces.

Establishment of the communications infrastructure begins by developing a system of interfaces that facilitate clean, organized data paths between subsystems. At the function level, the input/output (IO) interfaces are designated by the data being passed in and out of the function. This approach simply makes sense, as it is important for the user to identify the data that the function is working with. Less intuitive is the designation of interfaces at the subsystem level. Rather than pass individual signals between subsystems, the entire outputs of a particular function are merged

into a single signal and passed as subsystem-level data between subsystems. Thus, the interfaces at the subsystem level are designated according to the subsystems that they interact with. Figure 4-4 demonstrates this method of interface designation.

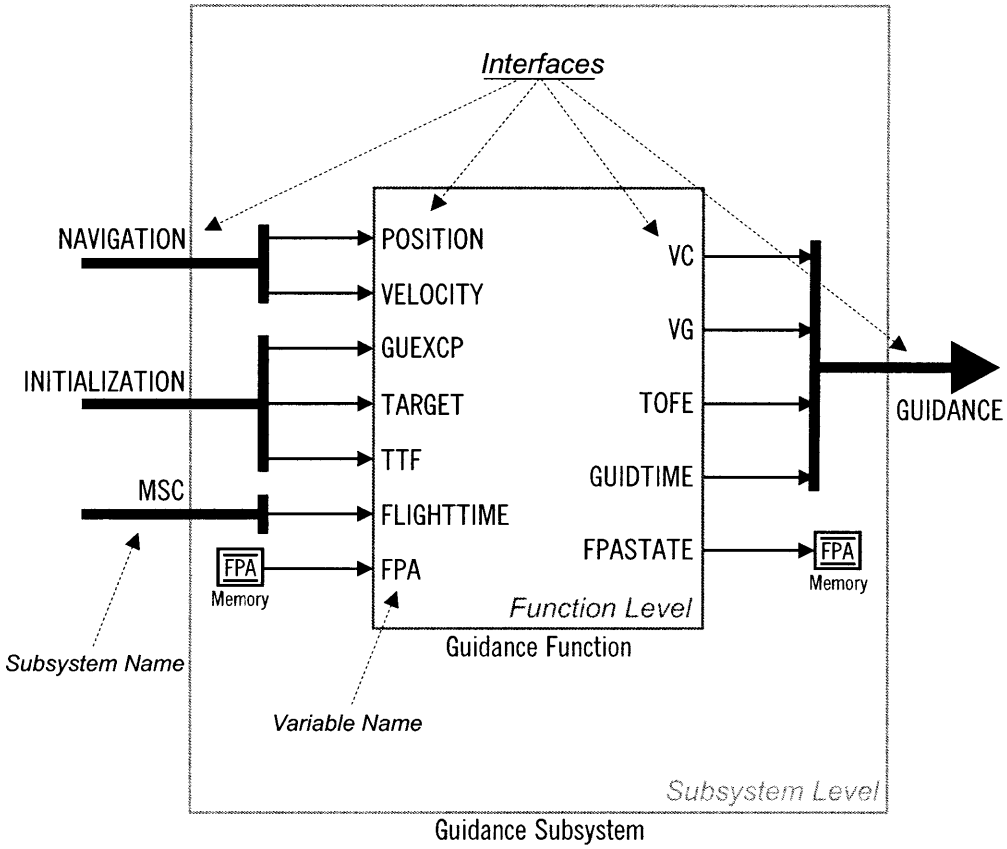

Figure 4-4: Interface Designation at the Function and Subsystem Levels

This method may seem trivial, but in reality it sets a subtle, effective precedent for minimization of subsystem interactions and system modifications. On the one hand, the system developer can readily identify from a system level the interactions of the various subsystems with each other. If a particular subsystem interacts with only one other subsystem, then perhaps it may be useful to merge the subsystems together. Conversely, if a subsystem interacts with an extraordinary number of other subsystems, it may serve the interests of interface minimization to break the subsystem up. On the other hand, a subsystem developer can view the function and subsystem interfaces and immediately trace specific data back to its origin without

having to look the data up in an interface control document (ICD). If the subsystem developer is considering modification or replacement of a function, he/she has immediate knowledge of the data that the function possesses instant access to. If the replacement function requires data not supplied by the subsystems already interfaced with the particular subsystem in question, then the developer is alerted that significant system modifications may be necessary to accomodate the new function.

The tracking of data passed between subsystems is augmented via a unique feature of Simulink called `bus signals`. The bus signal feature allows multiple signals to be named and hierarchically organized into a single larger signal for transport. The larger signal is named according to the parent function before being passed out of the subsystem interface. At the signal's destination, the individual signals can be extracted from the bus by name and connected to the appropriate function interfaces. This approach creates a well-organized system for tracking data and later evaluation of bandwidth requirements between subsystems.

Data Storage/Circumvention and Recovery

The establishment of circumvention and recovery routines requires identification of critical guidance system states. The framework for identifying these states actually derives from the method used to store persistent function variables from pass to pass. While it is not a stated requirement of the simulation, there is a desire to implement functions and components in the simulation with as little modification as possible. This presents a complication when attempting to integrate code for guidance functions in the simulation. When compiled language routines are interfaced with Simulink, they lose the capability to store data between passes. If storage of the data is required, the user must alter the code to include this data as a state of the model (Simulink models are updated via states ... see [6]). This requires a significant restructuring of the code in many cases.

In order to preserve the integrity of the code being used, an external method for storing function data must be implemented. The solution is to pass the data out of the function and store it in a local `memory block`. The memory block in Simulink is

basically a read/write mechanism that allows models to read and write from memory that is separate from the simulation's internal memory (states). Consequently, the critical data for each function is written to these memory blocks at the completion of a function pass, and then read into the function at the next pass. It turns out that this approach closely represents the storage of important data to hard memory in the guidance system. Figure 4-5 displays an example of a function's use of memory blocks to store critical data. Providing for data storage in this manner indirectly yields a significant advantage—the critical states of each function are automatically identified.

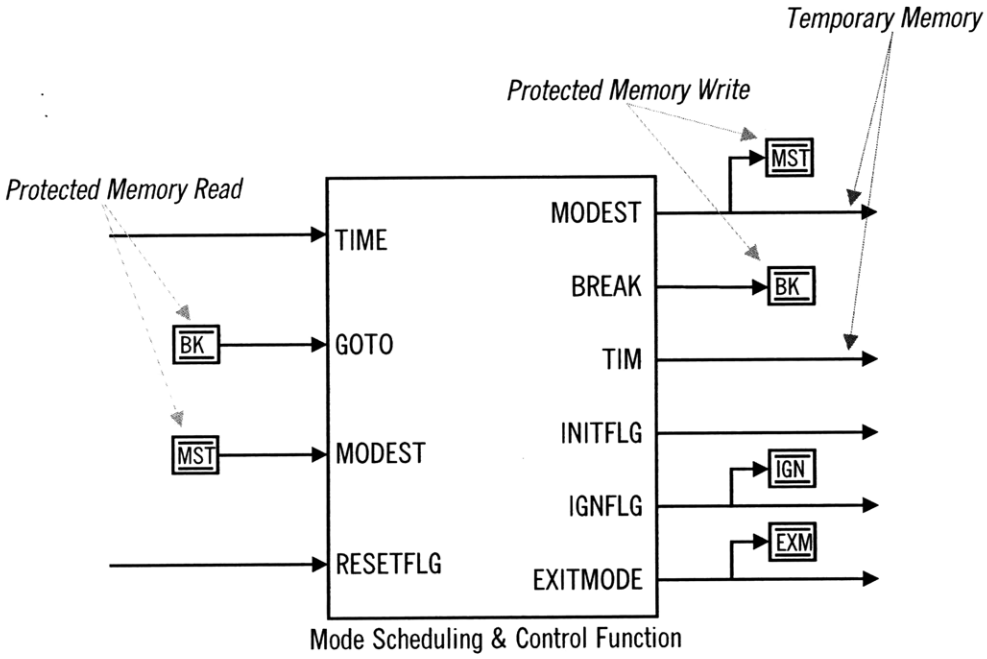

Figure 4-5: Function Data Storage

4.4 Modular Simulation Architectures

The goal of the simulation architectures developed in this thesis is to influence the modularity of the physical system as it is developed in simulation. Two target simulation architectures are established that could each provide unique perspectives on

the development of system modularity. Each facilitates a mode-based approach, and employs the same robust task scheduling, communications, and data handling frameworks developed in the previous section. The first architecture established is considered a horizontally partitioned architecture. The second is a vertically partitioned architecture. Each is established by laying out the guidance functions by mode (Figure 4-6), and then organizing the functions according to mode or function class.

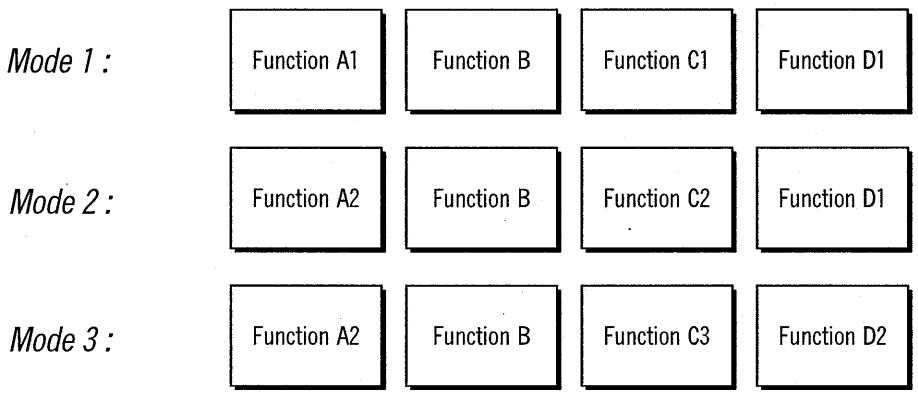

Figure 4-6: Functional Layout - By Mode

4.4.1 Horizontally Partitioned Architecture

Figure 4-7 shows the system-level organization of the horizontal architecture. The horizontal architecture organizes the guidance functions by mode. Each mode contains its own explicit set of functions and its own task scheduler to control the execution of those functions. If the same function(s) is executed in other modes, then a separate copy of that function exists in those modes as well. Knowledge of the specific mode of the system is only required by the top-level Mode Control system, which determines which mode should be activated at any given time.

4.4.2 Vertically Partitioned Architecture

The second architecture, referred to as the vertical architecture, is shown in Figure4-9). The guidance functions in this case are organized according to their appropriate

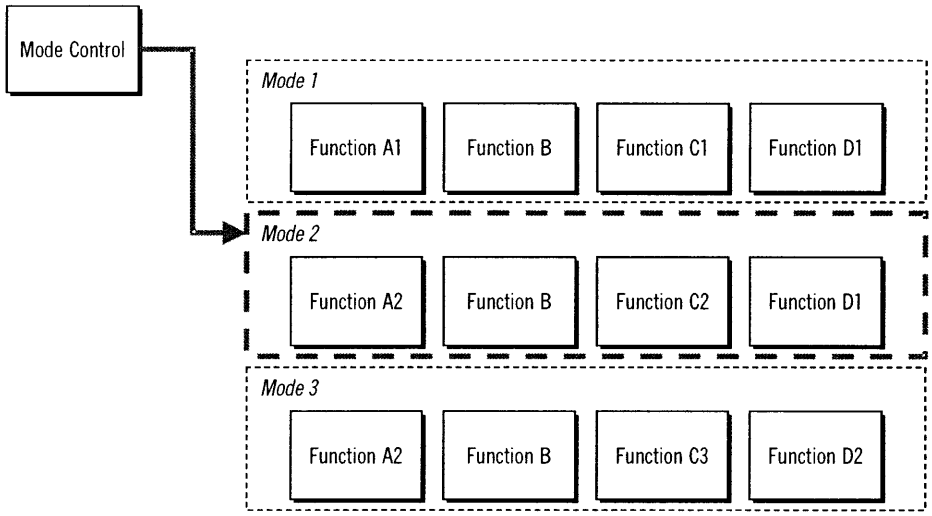

Figure 4-7: Horizontal Architecture

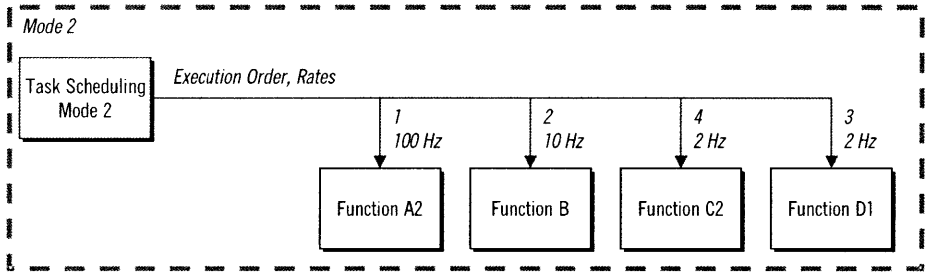

Figure 4-8: Horizontal Architecture - Task Scheduling

function class (i.e. all Steering routines grouped into a Steering class). Task scheduling routines are still separated by mode, but are contained with a single top-level scheduler. Instead of controlling the execution rate and order of specific functions, the function classes are executed by the top-level scheduler. Secondary logic within the function classes uses the system mode to determine which specific function to execute. Knowledge of system mode in this case is required by the task scheduler and the function classes.

Both architectures relegate determination of mode transition conditions to the individual MSC functions of each mode, and use a top-level Mode Control system to issue mode to the rest of the system. This type of mode control allows transition conditions and operational requirements within a particular mode to be altered

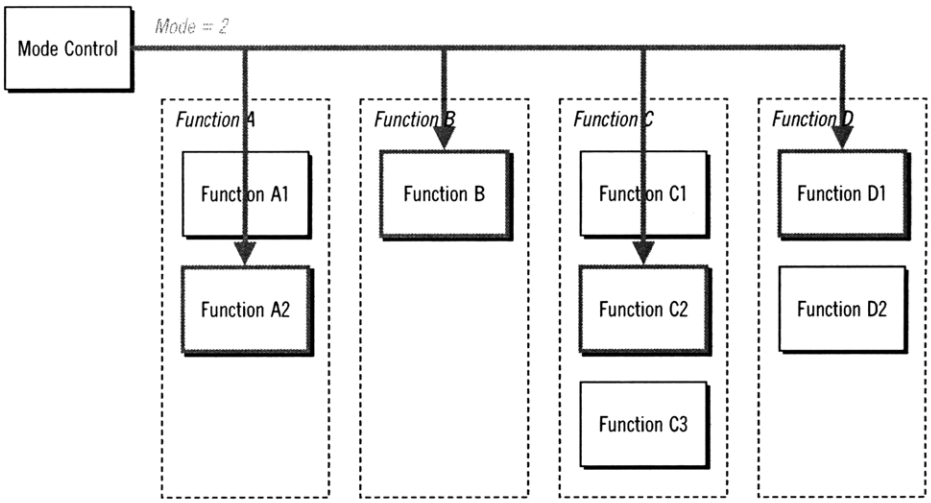

Figure 4-9: Vertical Architecture

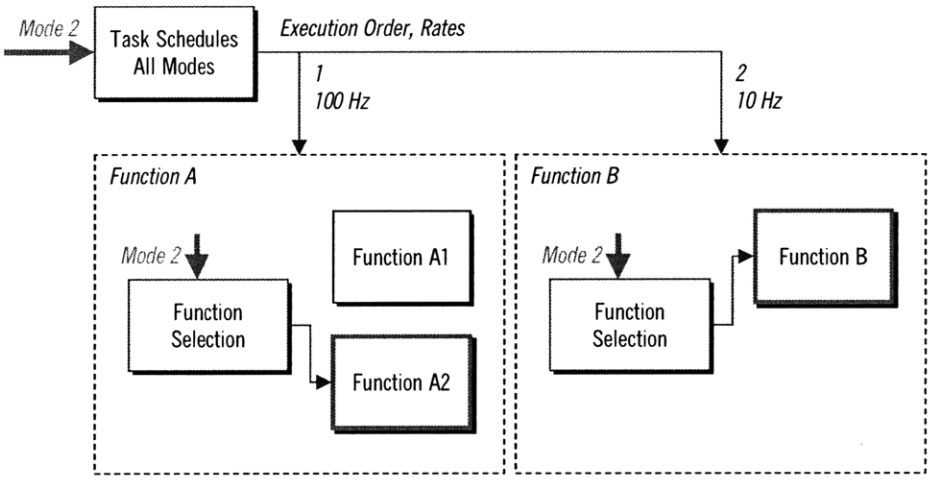

Figure 4-10: Vertical Architecture - Task Scheduling

without knowledge of or impact to the execution of other modes. Each architecture also employs the same interfaces at the function and subsystem level. In this way, the same functions can be integrated in either architecture without modification. This facilitates a more objective determination of which architecture supports more efficient insertion of alternate functions or functional re-partitioning.

Chapter 5

Simulation Validation

A robust framework has been established to facilitate analysis of critical guidance system design issues. That framework has been implemented in two modular simulation architectures developed with the intention of promoting system modularity. It is now time to validate the capabilities of the simulation(s) by using it to develop a generic strategic guidance system design. In addition, the usefulness of each architecture will be evaluated according to the simplicity with which each facilitates selected modular design studies.

5.1 Candidate System Design

The candidate system considered for development is a simplified, fully inertial guidance system. The functions and components for this guidance system have been designed entirely from scratch, to ensure a generalized study of simulation's capabilities and applications. The objective for validation of the simulation is to implement the candidate guidance system in each architecture, and establish that guidance system operation can be simulated through the boost phase[1] of flight. The design of the system will be considered successful by the verification of its ability to attain a specified correlated velocity at the end of boosted flight with no abnormal results.

[1]Development of the system for reentry-body deployment operations is not within the scope of this thesis.

5.1.1 Setup

The modes for this guidance system will be the same modes used as an example in Section 3.2, and included again in Table 5.1 for convenience. Part of the validation process will entail an evaluation of the simulation's ability to support the cycling of the system through these modes, with particular attention paid to the horizontal architecture where each mode is essentially its own self-contained system. Propagation of the system's data from one mode to the next could represent a substantial complication in the horizontal architecture.

Mode	Mission Event
0	Launch
1	1st Stage Burn
2	2nd Stage Burn
3	3rd Stage Burn
4	Post-Boost Overlap

Table 5.1: Candidate System Design Modes

The functions included in the system design are the primary functions described in Section 3.3 on page 27 and listed in Table 5.2. For the initial design, each mode of the guidance system will execute the same functions; an investigation of the insertion of alternate functions will take place later. Table 5.2 also defines the execution order (in descending order) and nominal rates for these functions, which will be coordinate via the task scheduling framework. A detailed listing of the code used for each of these functions can be found in Appendix A. Also for the initial design, the Vehicle Control function will be implemented as hardware and included as part of the missile model.

Completing the guidance system implementation is the missile model and Attitude/Velocity Measurement function. In order to evaluate guidance system performance, a three-stage missile model has been developed and implemented in the simulation (the model can be viewed in Appendix B). The missile model is the only continuous plant in the simulation. Providing 100-Hz sampled measurements of the missile's dynamics is the Attitude/Velocity Measurement function, which implements

Function	Rate
Mode Sequencing and Control	100-Hz
Initialization	10-Hz
Navigation	2-Hz
Guidance	2-Hz
Steering	2-Hz

Table 5.2: Candidate System Task Execution Order and Rates

a generic accelerometer model. The system possesses only a three degree-of-freedom capability, because the development of a 6-DOF capability would require simulation of a fully capable inertial measurement unit, requiring models for gyroscopes, gimbals, gimbal resolvers, platform control, etc. Since the objective of the system design is to validate that the simulation architectures work—not to actually design the next generation of strategic guidance systems—a 3-DOF simulation is quite sufficient.

5.1.2 Results

With the system modes established, the functions and components implemented, and all subsystems connected, the simulation can be executed. A complete hierarchical decomposition of the fully implemented system in the vertical architecture (only) can be found in Appendix C. The functions and components are implemented in the same manner in the horizontal architecture, with the exception of the top-level organization of the function subsystems. The simulation is executed for 80 seconds, using a variable-step solver, with an objective correlated velocity of 22,000 ft/sec. Graphs of the mode, missile stage, acceleration, and velocity on the following page (Figure 5-1) reveal that the simulation is a success. The guidance system cycles through the three missile stages (Modes 0-4), and attains the objective correlated velocity.

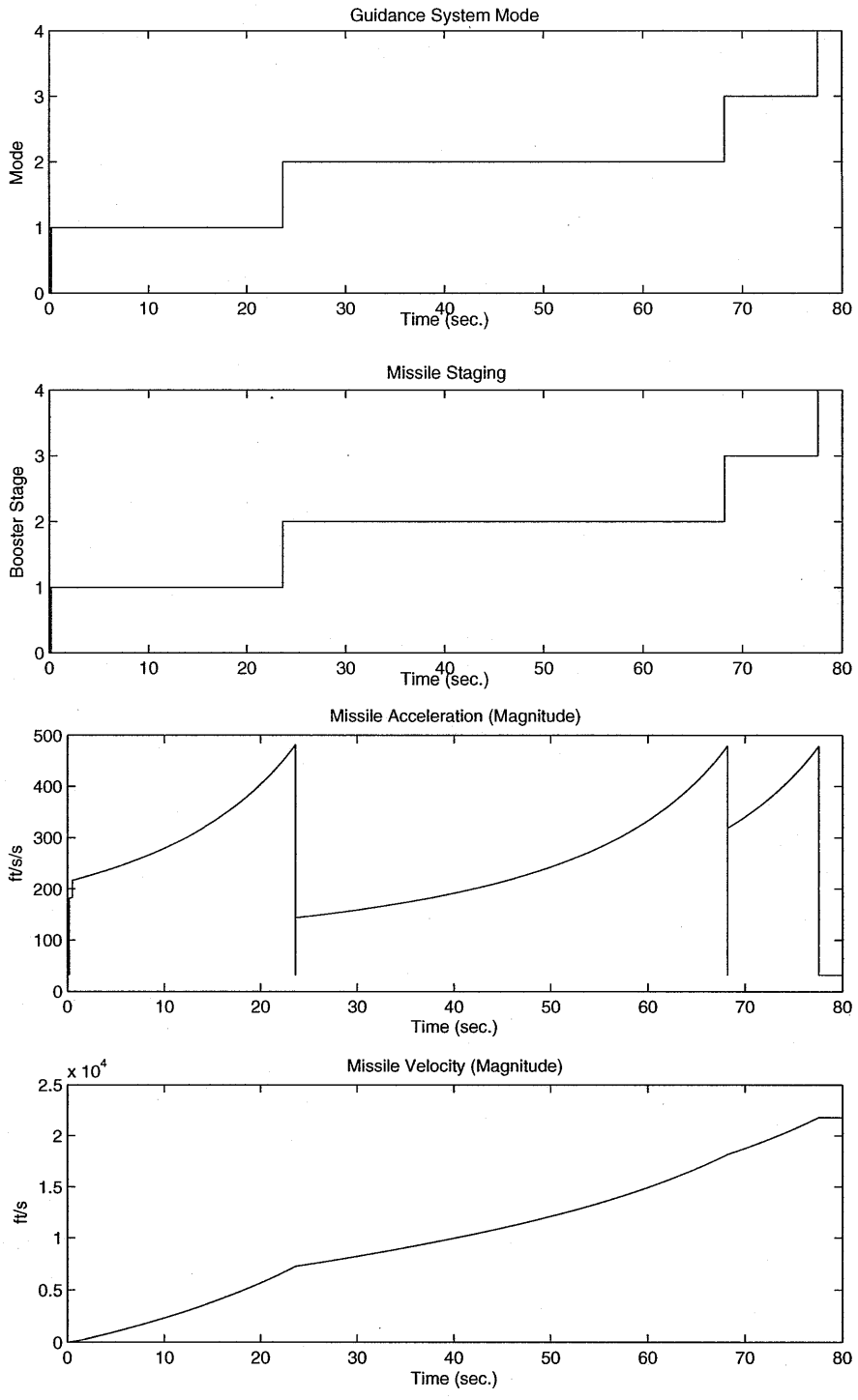

Figure 5-1: Simulation Results - Initial System Test

5.2 Evaluation of Architectures

Having verified that the simulation architectures can successfully support the simulation of a generic guidance system, the ability of the architectures to facilitate guidance system design studies must now be evaluated. Two modularity studies, traditionally difficult to simulate using previous simulation approaches, are examined with both architectures. In cases where one architecture yields significant advantages over the other, those advantages are pointed out. However, a final judgement of each architecture is withheld for the conclusions.

5.2.1 Modularity Studies

The simulation architectures are now tested for their ability to facilitate alternate configurations of system modularity studies. These tests include alteration of function execution rates as well as a functional repartitioning of the system. The objective is to capture the unique perspectives and advantages of each architecture with respect to these studies.

Alternate Task Scheduling

For this study, the plan is to increase the execution rate of the Navigation function to provide a more frequent update of the missile's position and velocity. The function's execution rate will be increased for all modes simulated. The objective is to discover which, if either, architecture facilitates the fewest system modifications to achieve this new execution rate, and to reveal any interesting behavior of the system in response to this new rate.

Neither architecture lends a particular advantage over the other for this study. Because the task scheduling routines are separated by mode in each architecture, the Navigation rate must be changed in each routine. In the horizontal architecture, this requires entering each mode's task scheduler and changing the execution rate. In the vertical architecture, the task scheduling routines are located in a single task scheduler, but each must still be brought up individually and changed.

However, this study does yield a significant result that leads to an improvement in system modularity. As stated earlier, the Navigation function's initial execution rate is 2-Hz. When the Navigation function was developed, it assumed that this rate would remain constant. Thus, the routine executed by Navigation accounts for the accelerometer's 100-Hz accumulated measurements of the change in velocity (ΔV), and assumes that the integration interval is then 0.5 seconds. When the execution rate of Navigation is changed to 5-Hz, the integration interval becomes 0.2 seconds and the missile's position and velocity is estimated incorrectly.

To solve this problem and provide for the ability to make future rate changes, the Navigation function is reengineered to assume no time interval. Instead, the function now stores the accumulated ΔV measurements and time interval, and uses the new measurements at each pass to calculated the elapsed interval for integration. The execution rate can now be changed—as well as the measurement rate of the accelerometer—without affecting Navigation calculations. The new Navigation function is included following the original function in Appendix A.

Alternate Functional Partitioning

The second study is an investigation of the architectures' ability to accomodate a repartitioning of the functions in the system. Assume that the guidance system design team wishes to remove the Vehicle Control function (presently run at 300-Hz) from hardware in the missile model and implement it as software in the guidance computer, where it will be executed at 100-Hz. The Vehicle Control function presently accepts an ignition flag from Mode Sequencing and Control (to initiate staging) and the boost vector from Guidance. It passes the boost vector and staging commands to the missile's interlocks assembly[2].

Relocation of the Vehicle Control function in the horizontal architecture proves to be an inefficient process. First, the Vehicle Control (VC) function is removed from the missile model, and the model's subsystem-level interface is changed to accept the boost and staging commands from outside the subsystem. The VC function must

[2]The interlocks assembly is the staging system of a typical strategic missile

then be implemented in each mode. This requires placing the VC function in the module and connecting it to the proper functions 5 times—once each for Modes 0-4. In addition, a function execution state must be added to the task scheduler in each mode; again, 5 times.

In the functional architecture, the same steps are taken to remove the VC function from the missile model. However, relocation of the function is much more efficient. A new function class is created for the VC function, and it is implemented and connected just one time. The task scheduling routines must still be modified individually.

The relocation of the VC function does not alter the system's performance. However, the relocation does improve the speed of the simulation; an obvious result of reducing the execution rate of the function.

[This page intentionally left blank.]

Chapter 6

Conclusions

6.1 Summary

This thesis has proposed, developed, and validated a top-down approach to the simulation-based design of strategic guidance systems. A robust simulation framework has been established to address key guidance system design considerations. That framework, along with a candidate system design, has been implemented in two modular simulation architectures that levy modular integration requirements on the subsystem designs. Together, these elements constitute a simulation concept that facilitates the development and analysis of system requirements in an integrated fashion.

The modular simulation architectures developed represent two different forms of functional partitioning: a horizontal method, where sets of functions are grouped according to the mode they are executed in (e.g. Mode 1 functions, Mode 3 functions, etc.); and a vertical method, where sets of functions are grouped according to the subsystem or function-class they belong to (e.g. Navigation functions, Steering functions, etc.). Each architecture represents a mode-based approach, but lends a different perspective to system design and interaction.

The horizontal architecture facilitates better identification and modification of system requirements by mode. A quick glance at the top level of functions immediately

reveals the specific functions executed within that mode. However, the horizontal architecture is more difficult to work with when making system modifications, as a single modification must be made to multiple occurrences of the function or component. The vertical architecture provides more flexibility in this respect. Alternate functions can be rapidly inserted into the system and set up for execution in a particular mode(s). In addition, functions occur only once in the system, which simplifies modifications. The vertical architecture more accurately represents the organization of functions in a physical system.

6.2 Future Work

The scope of this thesis was necessarily limited to the development of the simulation approach and its validation. Now that this approach has proved successful, full population of the guidance system can occur. The modular simulation architectures can support the insertion of any number of models in any of the existing subsystems, or the insertion of additional subsystems without extensive modification of the simulation framework. The full set of primary, implementing, and support functions, in addition to the appropriate insrument, communications, and missile system models, should be implemented in the simulations.

Additional simulation architectures, representing alternate forms of partitioning (physical, computational) or different aspects of the guidance system design (power, thermal, etc.) can also be constructed using the same approach. Rather than partitioning the functions according mode or class, the functions could be partitioned according to the processor or hardware that they reside on. A simulation could be constructed that represents the power or thermal subsystem alone, and then integrated as a separate layer of the overall system simulation.

Appendix A

Candidate System Functions

This appendix lists the functions used in the candidate guidance system simulation described in Chapter 5. The code for these functions is written and compiled in standard ANSI C, and implemented in Simulink as `S-Functions`. S-Functions are the means by which Simulink is able to incorporate compiled languages. In each case, the wrapper code required by Simulink to interface the functions with the graphical interface has been left out.

A.1 Mode Scheduling & Control

The Mode Scheduling and Control functions implemented in the simulation perform the same essential actions from mode to mode. Each function tracks the time elapsed in the present mode and checks this elapsed time against a pre-loaded test time used to ensure that each stage is allowed to burn for a sufficient length of time before igniting the next stage. Once a sufficient amount of time has elapsed, the MSC function sends a flag (IGNFLG = X1, where X is the stage number) to the Vehicle Control function issuing permission to ignite the next stage when the current stage is exhausted. Upon ignition of the next stage, Vehicle Control sends a reply to MSC (RESETFLG = 1), declaring that the next stage has been successfully ignited. MSC acknowledges this reply by decrementing the IGNFLG (IGNFLG = X0), and the VC function responds

in kind by resetting the RESETFLG (RESETFLG = 0). MSC then issues the mode exit (EXITMODE = 1) flag to Mode Transition and Control to signal that the next mode can be entered.

In addition to supervision of missile staging, the MSC function issues and necessary initialization commands at the beginning of each mode. The breakpoints found throughout the code are used by MSC to track its progress through each particular mode.

A.1.1 Mode 0 Scheduling & Control Code

```
// Map Input Ports
const real_T *time              = ssGetInputPortSignal(S,0);
const real_T *breakpoint_in     = ssGetInputPortSignal(S,1);
const real_T *modest_in         = ssGetInputPortSignal(S,2);
const real_T *resetflg          = ssGetInputPortSignal(S,3);

// Map Output Ports
real_T *modest          = ssGetOutputPortSignal(S,0);
real_T *breakpoint      = ssGetOutputPortSignal(S,1);
real_T *tim             = ssGetOutputPortSignal(S,2);
real_T *initflg         = ssGetOutputPortSignal(S,3);
real_T *ignflg          = ssGetOutputPortSignal(S,4);
real_T *exitmode        = ssGetOutputPortSignal(S,5);

// Load Constants
double ttest    = 0;

// Update Time-in-Mode
*tim    = *time - *modest_in;

// Check if this is the first run...
if (*breakpoint_in == 0.0)
{
        // Initialize Mode
        *initflg        = 1;            // Perform any initialization routines
        *modest         = *time;        // Save Mode Start Time
        *tim            = 0;            // Reset Time-in-Mode
        *exitmode       = 0;            // Reset EXITMODE
        *ignflg         = 10;           // Initialize IGNFLG
        *breakpoint     = 0.1;          // Set breakpoint for next pass
}
else if (*breakpoint_in == 0.1)
{
        // Check Time-in-Mode
        if (*tim > ttest)
        {
                // Enable Interlocks Permissive
                *ignflg     = 11;
                *breakpoint = 0.2;
        }
}
else if (*breakpoint_in == 0.2)
{
        // Check for Reset from VC
        if (*resetflg == 1)
        {
                // Confirm Reset
                *ignflg         = 10;
                *breakpoint     = 0.3;
        }
}
else if (*breakpoint == 0.3)
{
        // Check for Acknowledgement from VC
        if (*resetflg == 0)
        {
                // Transition to Mode 1
                *exitmode       = 1;
                *breakpoint     = 1.0;
        }
}
else
{
        ;
}
```

A.1.2 Mode 1 Scheduling & Control Code

```
// Map Input Ports
const real_T *time              = ssGetInputPortSignal(S,0);
const real_T *breakpoint_in     = ssGetInputPortSignal(S,1);
const real_T *modest_in         = ssGetInputPortSignal(S,2);
const real_T *resetflg          = ssGetInputPortSignal(S,3);

// Map Output Ports
real_T *modest          = ssGetOutputPortSignal(S,0);
real_T *breakpoint      = ssGetOutputPortSignal(S,1);
real_T *tim             = ssGetOutputPortSignal(S,2);
real_T *initflg         = ssGetOutputPortSignal(S,3);
real_T *ignflg          = ssGetOutputPortSignal(S,4);
real_T *exitmode        = ssGetOutputPortSignal(S,5);

// Load Constants
double ttest    = 20;

// Update Time-in-Mode
*tim    = *time - *modest_in;

// Check if this is the first run...
if (*breakpoint_in == 1.0)
{
        // Initialize Mode
        *modest         = *time;        // Save MODEST
        *tim            = 0;            // Reset Time-in-Mode
        *exitmode       = 0;            // Reset EXITMODE
        *ignflg         = 10;
        *breakpoint     = 1.1;
}
else if (*breakpoint_in == 1.1)
{
        // Check Time-in-Mode
        if (*tim > ttest)
        {
                // Enable Interlocks Permissive
                *ignflg     = 21;
                *breakpoint = 1.2;
        }
}
else if (*breakpoint_in == 1.2)
{
        // Check for Reset from VC
        if (*resetflg == 1)
        {
                // Confirm Reset
                *ignflg         = 20;
                *breakpoint     = 1.3;
        }
}
else if (*breakpoint == 1.3)
{
        // Check for Acknowledgement from VC
        if (*resetflg == 0)
        {
                // Transition to Mode 2
                *exitmode       = 1;
                *breakpoint     = 2.0;
        }
}
else
{
        ;
}
```

A.1.3 Mode 2 Scheduling & Control Code

```
// Map Input Ports
const real_T *time              = ssGetInputPortSignal(S,0);
const real_T *breakpoint_in     = ssGetInputPortSignal(S,1);
const real_T *modest_in         = ssGetInputPortSignal(S,2);
const real_T *resetflg          = ssGetInputPortSignal(S,3);

// Map Output Ports
real_T *modest          = ssGetOutputPortSignal(S,0);
real_T *breakpoint      = ssGetOutputPortSignal(S,1);
real_T *tim             = ssGetOutputPortSignal(S,2);
real_T *initflg         = ssGetOutputPortSignal(S,3);
real_T *ignflg          = ssGetOutputPortSignal(S,4);
real_T *exitmode        = ssGetOutputPortSignal(S,5);

// Load Constants
double ttest    = 40;

// Update Time-in-Mode
*tim    = *time - *modest_in;

// Check if this is the first run...
if (*breakpoint_in == 2.0)
{
        // Initialize Mode
        *modest                 = *time;        // Save MODEST
        *tim                    = 0;            // Reset Time-in-Mode
        *exitmode               = 0;            // Reset EXITMODE
        *ignflg                 = 20;
        *breakpoint             = 2.1;
}
else if (*breakpoint_in == 2.1)
{
        // Check Time-in-Mode
        if (*tim > ttest)
        {
                // Enable Interlocks Permissive
                *ignflg     = 31;
                *breakpoint = 2.2;
        }
}
else if (*breakpoint_in == 2.2)
{
        // Check for Reset from VC
        if (*resetflg == 1)
        {
                // Confirm Reset
                *ignflg         = 30;
                *breakpoint     = 2.3;
        }
}
else if (*breakpoint == 2.3)
{
        // Check for Acknowledgement from VC
        if (*resetflg == 0)
        {
                // Transition to Mode 3
                *exitmode       = 1;
                *breakpoint     = 3.0;
        }
}
else
{
        ;
}
```

A.1.4 Mode 3 Scheduling & Control Code

```
// Map Input Ports
const real_T *time              = ssGetInputPortSignal(S,0);
const real_T *breakpoint_in     = ssGetInputPortSignal(S,1);
const real_T *modest_in         = ssGetInputPortSignal(S,2);
const real_T *resetflg          = ssGetInputPortSignal(S,3);

// Map Output Ports
real_T *modest          = ssGetOutputPortSignal(S,0);
real_T *breakpoint      = ssGetOutputPortSignal(S,1);
real_T *tim             = ssGetOutputPortSignal(S,2);
real_T *initflg         = ssGetOutputPortSignal(S,3);
real_T *ignflg          = ssGetOutputPortSignal(S,4);
real_T *exitmode        = ssGetOutputPortSignal(S,5);

// Load Constants
double ttest    = 5;

// Update Time-in-Mode
*tim    = *time - *modest_in;

// Check if this is the first run...
if (*breakpoint_in == 3.0)
{
        // Initialize Mode
        *modest                 = *time;        // Save MODEST
        *tim                    = 0;            // Reset Time-in-Mode
        *exitmode               = 0;            // Reset EXITMODE
        *ignflg                 = 30;
        *breakpoint             = 3.1;
}
else if (*breakpoint_in == 3.1)
{
        // Check Time-in-Mode
        if (*tim > ttest)
        {
                // Enable Interlocks Permissive
                *ignflg     = 41;
                *breakpoint = 3.2;
        }
}
else if (*breakpoint_in == 3.2)
{
        // Check for Reset from VC
        if (*resetflg == 1)
        {
                // Confirm Reset
                *ignflg         = 40;
                *breakpoint     = 3.3;
        }
}
else if (*breakpoint == 3.3)
{
        // Check for Acknowledgement from VC
        if (*resetflg == 0)
        {
                // Transition to Mode 4
                *exitmode       = 1;
                *breakpoint     = 4.0;
        }
}
else
{
        ;
}
```

A.1.5 Mode 4 Scheduling & Control Code

```
// Map Input Ports
const real_T *time              = ssGetInputPortSignal(S,0);
const real_T *breakpoint_in     = ssGetInputPortSignal(S,1);
const real_T *modest_in         = ssGetInputPortSignal(S,2);
const real_T *resetflg          = ssGetInputPortSignal(S,3);

// Map Output Ports
real_T *modest          = ssGetOutputPortSignal(S,0);
real_T *breakpoint      = ssGetOutputPortSignal(S,1);
real_T *tim             = ssGetOutputPortSignal(S,2);
real_T *initflg         = ssGetOutputPortSignal(S,3);
real_T *ignflg          = ssGetOutputPortSignal(S,4);
real_T *exitmode        = ssGetOutputPortSignal(S,5);

// Load Constants
double ttest    = 0;

// Update Time-in-Mode
*tim    = *time - *modest_in;

// Check if this is the first run...
if (*breakpoint_in == 4.0)
{
        // Initialize Mode
        *modest                 = *time;        // Save MODEST
        *tim                    = 0;            // Reset Time-in-Mode
        *exitmode               = 0;            // Reset EXITMODE
        *ignflg                 = 40;
        *breakpoint             = 4.1;
}
else if (*breakpoint_in == 4.1)
{
;
}
else
{
        ;
}
```

A.2 Initialization

The Initialization function accepts flags from MSC designating appropriate initialization actions for each mode (only Mode 0 requires initialization in this case). For instance, an INITFLG of value 1 causes the Initialization function to signal Navigation to use the launch conditions (NAVSNAP) supplied by Initialization (rather than the last calculated conditions, which are absent at launch), and to signal Guidance to not issue steering commands (GUEXCP) until a stage has been ignited. The Initialization function in this case also supplies the target loads which are used by Guidance to calculate the correlated velocity.

A.2.1 Initialization Code

```
// Map Input Ports
const real_T *initflg   = ssGetInputPortSignal(S,0);
const real_T *tindex_in = ssGetInputPortSignal(S,1);
const real_T *dataload  = ssGetInputPortSignal(S,2);

// Map Output Ports
real_T *guexcp          = ssGetOutputPortSignal(S,0);
real_T *infpa           = ssGetOutputPortSignal(S,1);
real_T *navsnap         = ssGetOutputPortSignal(S,2);
real_T *rm              = ssGetOutputPortSignal(S,3);
real_T *vm              = ssGetOutputPortSignal(S,4);
real_T *tindex_out      = ssGetOutputPortSignal(S,5);
real_T *rt              = ssGetOutputPortSignal(S,6);
real_T *ttf             = ssGetOutputPortSignal(S,7);

// Calculate Outputs

if (*initflg == 1)
{
        *guexcp  = 1;
        *navsnap = 1;
}
else
{
        *guexcp  = 0;
        *navsnap = 0;
}

// Read in Launch Position and Velocity to Nav
rm[0]   = dataload[0];
rm[1]   = dataload[1];
rm[2]   = dataload[2];

vm[0]   = dataload[3];
vm[1]   = dataload[4];
vm[2]   = dataload[5];

// Select Target
if (*tindex_in == 1)
{
        rt[0]   = dataload[7];
        rt[1]   = dataload[8];
        rt[2]   = dataload[9];
        *ttf    = dataload[6];
}
else if (*tindex_in == 2)
{
        rt[0]   = dataload[11];
        rt[1]   = dataload[12];
        rt[2]   = dataload[13];
        *ttf    = dataload[10];
}
else if (*tindex_in == 3)
{
        rt[0]   = dataload[15];
        rt[1]   = dataload[16];
        rt[2]   = dataload[17];
        *ttf    = dataload[14];
}
else
{
;
}

// Read in Guidance initial FPA guess
*infpa = dataload[18];
```

```
*tindex_out      = *tindex_in;
```

A.3 Navigation

The Navigation function here calculates the missile's current position and velocity using a trapezoidal integration method and a mid-point estimate of spherical gravity. As discussed in Chapter 5, this function has been modified from its original form to accept the accumulated "ΔV" measurements from Velocity Measurement at any time interval.

A.3.1 Navigation Code

```
// Map Input Ports
const real_T *delta_v   = ssGetInputPortSignal(S,0);
const real_T *dvlast    = ssGetInputPortSignal(S,1);
const real_T *navsnap   = ssGetInputPortSignal(S,2);
const real_T *rlaunch   = ssGetInputPortSignal(S,3);
const real_T *vlaunch   = ssGetInputPortSignal(S,4);
const real_T *rmlast    = ssGetInputPortSignal(S,5);
const real_T *vmlast    = ssGetInputPortSignal(S,6);

// Map Output Ports
real_T *rm              = ssGetOutputPortSignal(S,0);
real_T *vm              = ssGetOutputPortSignal(S,1);
real_T *dvstore         = ssGetOutputPortSignal(S,2);

// Initialize Internal Variable
double dt               = 0;     // time increment of delta_V
double mu               = 0;     // gravitational constant
double rm_mdpt_mag      = 0;     // rm_mdpt magnitude

double rmo[3]           = {0};   // position initial condition
double vmo[3]           = {0};   // velocity initial condition
double dv[3]            = {0};   // delta v for calculations
double gm[3]            = {0};   // gravity vector (estimated)
double rm_mdpt[3]       = {0};   // rm_mdpt (estimated)
double rm_mdpt_unit[3]  = {0};   // rm_mdpt direction

// Calculate Outputs
mu = 1.407646882*pow(10,16);     // (ft^3/s^2)

// Compute time interval
dt = delta_v[0] - dvlast[0];

// Compute dV accumulation
dv[0]   = delta_v[1] - dvlast[1];
dv[1]   = delta_v[2] - dvlast[2];
dv[2]   = delta_v[3] - dvlast[3];

// Read initial conditions
if (*navsnap == 1)
{
        // Use launch conditions
        rmo[0] = rlaunch[0];
        rmo[1] = rlaunch[1];
        rmo[2] = rlaunch[2];

        vmo[0] = vlaunch[0];
        vmo[1] = vlaunch[1];
        vmo[2] = vlaunch[2];
}
else
{
        // Use last calculated conditions
        rmo[0] = rmlast[0];
        rmo[1] = rmlast[1];
        rmo[2] = rmlast[2];

        vmo[0] = vmlast[0];
        vmo[1] = vmlast[1];
        vmo[2] = vmlast[2];
}

// Estimate mid-point position
rm_mdpt[0] = rmo[0] + vmo[0]*(dt/2);
rm_mdpt[1] = rmo[1] + vmo[1]*(dt/2);
rm_mdpt[2] = rmo[2] + vmo[2]*(dt/2);
```

```
rm_mdpt_mag = sqrt(pow(rm_mdpt[0],2) + pow(rm_mdpt[1],2) + pow(rm_mdpt[2],2));

rm_mdpt_unit[0] = rm_mdpt[0]/rm_mdpt_mag;
rm_mdpt_unit[1] = rm_mdpt[1]/rm_mdpt_mag;
rm_mdpt_unit[2] = rm_mdpt[2]/rm_mdpt_mag;

// Compute mid-point gravity vector
gm[0] = -(mu/pow(rm_mdpt_mag,2))*rm_mdpt_unit[0];
gm[1] = -(mu/pow(rm_mdpt_mag,2))*rm_mdpt_unit[1];
gm[2] = -(mu/pow(rm_mdpt_mag,2))*rm_mdpt_unit[2];

// Compute current missile velocity
vm[0] = dv[0] + gm[0]*dt + vmo[0];
vm[1] = dv[1] + gm[1]*dt + vmo[1];
vm[2] = dv[2] + gm[2]*dt + vmo[2];

// Compute current missile position
rm[0] = (vm[0] + vmo[0])*dt/2 + rmo[0];
rm[1] = (vm[1] + vmo[1])*dt/2 + rmo[1];
rm[2] = (vm[2] + vmo[2])*dt/2 + rmo[2];

// Store delta v from this pass
dvstore[0]      = delta_v[0];
dvstore[1]      = delta_v[1];
dvstore[2]      = delta_v[2];
dvstore[3]      = delta_v[3];
```

A.4 Guidance

This Guidance function uses a general Lambert guidance algorithm extracted from pp. 281-291 of Zarchan [8]. The function arrives at a solution for the correlated velocity by iterating on the flight path angle and matching the resulting time-of-flight with the time-of-flight desired (the remaining time left in flight). The correlated velocity is calculated in the missile-target plane (the Lambert frame) and transformed back to the inertial frame.

A.4.1 Guidance Code

```
// Map Input Ports
const real_T *rm        = ssGetInputPortSignal(S,0);
const real_T *vm        = ssGetInputPortSignal(S,1);
const real_T *guexcp    = ssGetInputPortSignal(S,2);
const real_T *infpa     = ssGetInputPortSignal(S,3);
const real_T *rt        = ssGetInputPortSignal(S,4);
const real_T *ttf       = ssGetInputPortSignal(S,5);
const real_T *sys_time  = ssGetInputPortSignal(S,6);
const real_T *gamma_old = ssGetInputPortSignal(S,7);

// Map Output Ports
real_T *vc              = ssGetOutputPortSignal(S,0);
real_T *vg              = ssGetOutputPortSignal(S,1);
real_T *tof_elps        = ssGetOutputPortSignal(S,2);
real_T *gu_time         = ssGetOutputPortSignal(S,3);
real_T *gamma           = ssGetOutputPortSignal(S,4);

// Initialize Internal Variables
double i                = 0;                    // iteration counter
double t                = 0;                    // simulation time
double mu               = 1.407646882*pow(10,16); // gravitational constant
double guess_flag       = 0;    // flag for initial guess on first guidance pass

double tof_des          = 0;    // remaining time-of-flight desired

double rm_mag           = 0;    // magnitude of missile position vector
double rt_mag           = 0;    // magnitude of target position vector

double un_mag           = 0;    // magnitude of un_li (to form unit vector)
double ut_mag           = 0;    // magnitude of ut_li (to form unit vector)

double un_li[3] = {0};  // target/missile orthogonal unit vector
double ur_li[3] = {0};  // missile position unit vector
double ut_li[3] = {0};  // orthogonal unit vector to un_li and ur_li
                        // (all for Lambert->Inertial Transformation)

double T_li[3][3]       = {0};  // Lambert->Inertial Transformation Matrix

double phi              = 0;  // central range angle
double gmintop          = 0;  // minimum f.p.a. calculation - numerator
double gmaxtop          = 0;  // maximum f.p.a. calculation - numerator
double gbottom          = 0;  // f.p.a calculation - denominator
double gmin             = 0;  // minimum f.p.a.
double gmax             = 0;  // maximum f.p.a.
double gamma_n          = 0;  // f.p.a. for nth iteration
double gamma_last       = 0;  // f.p.a. for n-1 iteration
double gamma_next       = 0;  // f.p.a. for n+1 iteration

double term1            = 0;  // 1st term of Lambert velocity calculation
double term2            = 0;  // 2nd ...
double term3            = 0;  // 3rd ...
double term4            = 0;  // 4th ...
double v_lamb           = 0;  // Lambert velocity
double vl[3]            = 0;  // Lambert velocity vector

double lambda           = 0;  // ballistic constant

double tterm1           = 0;  // 1st component of time-of-flight calculation
double tterm2           = 0;  // 2nd ...
double tterm3           = 0;  // 3rd ...
double tterm4           = 0;  // 4th ...
double tterm5           = 0;  // 5th ...
double tof              = 0;  // time-of-flight for pass n
double tof_last         = 0;  // time-of-flight from pass n-1

t = ssGetT(S);
```

```
// Begin Calculations
// Check for initialization
if (*guexcp == 1)
{
        *gu_time        = *sys_time;
        *tof_elps       = 0;
        *gamma          = *infpa;

        vc[0] = 0;
        vc[1] = 0;
        vc[2] = 0;

        vg[0] = rm[0];
        vg[1] = rm[1];
        vg[2] = rm[2];

        guess_flag = 1;

}
// Proceed as normal
else
{
        // Augment guidance time
        *tof_elps = *tof_elps + (*sys_time - *gu_time);

        // Calculate remaining tof desired
        tof_des = *ttf - *tof_elps;

        // Compute magnitudes of missile and target vector
        rm_mag = sqrt(pow(rm[0],2) + pow(rm[1],2) + pow(rm[2],2));
        rt_mag = sqrt(pow(rt[0],2) + pow(rt[1],2) + pow(rt[2],2));

        // Compute Lambert->Inertial Transformation
        // un_li : cross(rm, rt)
        un_li[0] = rm[1]*rt[2] - rm[2]*rt[1];
        un_li[1] = rm[2]*rt[0] - rm[0]*rt[2];
        un_li[2] = rm[0]*rt[1] - rm[1]*rt[0];

        un_mag = sqrt(pow(un_li[0],2) + pow(un_li[1],2) + pow(un_li[2],2));

        un_li[0] = un_li[0]/un_mag;
        un_li[1] = un_li[1]/un_mag;
        un_li[2] = un_li[2]/un_mag;

        // ur_li
        ur_li[0] = rm[0]/rm_mag;
        ur_li[1] = rm[1]/rm_mag;
        ur_li[2] = rm[2]/rm_mag;

        // ut_li : cross(un_li, ur_li)
        ut_li[0] = un_li[1]*ur_li[2] - un_li[2]*ur_li[1];
        ut_li[1] = un_li[2]*ur_li[0] - un_li[0]*ur_li[2];
        ut_li[2] = un_li[0]*ur_li[1] - un_li[1]*ur_li[0];

        ut_mag = sqrt(pow(ut_li[0],2) + pow(ut_li[1],2) + pow(ut_li[2],2));

        ut_li[0] = ut_li[0]/ut_mag;
        ut_li[1] = ut_li[1]/ut_mag;
        ut_li[2] = ut_li[2]/ut_mag;

        // Transformation
        T_li[0][0] = ut_li[0];
        T_li[0][1] = ur_li[0];
        T_li[0][2] = un_li[0];
        T_li[1][0] = ut_li[1];
        T_li[1][1] = ur_li[1];
        T_li[1][2] = un_li[1];
```

```
T_li[2][0] = ut_li[2];
T_li[2][1] = ur_li[2];
T_li[2][2] = un_li[2];

// Compute central range angle
phi = acos((rm[0]*rt[0] + rm[1]*rt[1] + rm[2]*rt[2])/(rm_mag*rt_mag));

// Compute min/max f.p.a. for central range angle
gmintop = sin(phi) - sqrt(2*(rm_mag/rt_mag)*(1-cos(phi)));
gmaxtop = sin(phi) + sqrt(2*(rm_mag/rt_mag)*(1-cos(phi)));
gbottom = 1 - cos(phi);

gmin    = atan2(gmintop,gbottom);

gmax    = atan2(gmaxtop,gbottom);

// Use initial guess if this is first pass
if (guess_flag == 1)
{
        gamma_n = (gmin + gmax)/2;

        guess_flag = 0;
}
else    // use last flight path angle as new guess
{
        gamma_n = *gamma_old;
}

// Iterate to find Lambert f.p.a.
for (i = 0; i < 10; i++)
{

        // Compute Lambert velocity
        term1 = mu*(1 - cos(phi));
        term2 = (rm_mag*cos(gamma_n));
        term3 = (rm_mag*cos(gamma_n)/rt_mag);
        term4 = cos(phi + gamma_n);

        v_lamb = sqrt(term1/(term2*(term3 - term4)));

        // Compute ballistic coefficient
        lambda = (rm_mag*pow(v_lamb,2))/mu;

        // Check if current v_lamb, gamma satisfy time-of-flight
        tterm1 = tan(gamma_n)*(1 - cos(phi)) + (1 - lambda)*sin(phi);
        tterm2 = ((2 - gamma_n)*(1 - cos(phi)));
        tterm3 = lambda*pow(cos(gamma_n),2) + cos(gamma_n + phi)/cos(gamma_n);
        tterm4 = lambda*pow(2/lambda - 1, 1.5);
        tterm5 = sqrt(2/lambda - 1)/(cos(gamma_n)*(1/tan(phi/2)) - sin(gamma_n));

        tof = (rm_mag/(v_lamb*cos(gamma_n)))*(tterm1/(tterm2/tterm3) + ...
              ... (2*cos(gamma_n)/tterm4)*atan(tterm5));

        // Check tof against tof_des
        if (abs(tof - tof_des) > .0001)
        {
                if (tof > tof_des)
                {
                        gmax = gamma_n;
                }
                else
                {
                        gmin = gamma_n;
                }

                // If first iteration, initial guess
                if (i == 0)
                {
```

```
                        // Calculate gamma (n+1)
                        gamma_next = (gmax + gmin)/2;
                }
                else
                {
                        // Use value from previous iteration
                        // Calculate gamma (n+1)
                        gamma_next = gamma_n + (gamma_n - gamma_last)*(tof_des - tof)/(tof - tof_last);

                        if ((gamma_next > gmax) || (gamma_next < gmin))
                        {
                                gamma_next = (gmax + gmin)/2;
                        }

                }

                // Update values for next iteration
                gamma_last      = gamma_n;
                tof_last        = tof;
                gamma_n         = gamma_next;
        }
        else
        {
                // We have a good solution - no more iterations required
                break;
        }

}

// Compute Lambert velocity
vl[0] = v_lamb*cos(gamma_n);
vl[1] = v_lamb*sin(gamma_n);
vl[2] = 0;

// Compute correlated velocity
vc[0] = T_li[0][0]*vl[0] + T_li[0][1]*vl[1] + T_li[0][2]*vl[2];
vc[1] = T_li[1][0]*vl[0] + T_li[1][1]*vl[1] + T_li[1][2]*vl[2];
vc[2] = T_li[2][0]*vl[0] + T_li[2][1]*vl[2] + T_li[2][2]*vl[2];

// Compute velocity-to-be-gained
vg[0] = vc[0] - vm[0];
vg[1] = vc[1] - vm[1];
vg[2] = vc[2] - vm[2];

// Pass out current flight path angle
*gamma = gamma_n;

// Pass out guidance time of calculation
*gu_time        = *sys_time;

}
```

A.5 Steering

The Steering function here uses a simple $\vec{V_g}$ routine—it simply seeks to null the difference between the missile velocity and the correlated velocity by continuously following the velocity-to-be-gained.

A.5.1 Steering Code

```
// Map Input Ports
const real_T *vg = ssGetInputPortSignal(S,0);

// Map Output Ports
real_T *boost = ssGetOutputPortSignal(S,0);

// Initialize Internal Variables
double vg_mag = 0;   // magnitude of velocity-to-be-gained

// Calculate Thrust Vector for General Steering
vg_mag  = sqrt(pow(vg[0],2) + pow(vg[1],2) + pow(vg[2],2));

boost[0] = vg[0]/vg_mag;
boost[1] = vg[1]/vg_mag;
boost[2] = vg[2]/vg_mag;
```

A.6 Vehicle Control

The Vehicle Control function interacts with the MSC and Interlocks functions to coordinate operation of the missile's propulsion system. When given permission by MSC, the Vehicle Control function monitors the missile's acceleration due to thrust (and the missile's velocity during launch). When the acceleration drops below a certain threshold, the VC function initiates ignition of the next stage. Staging commands are sent to the Interlocks function, which controls the activation of the appropriate missile stages.

A.6.1 Vehicle Control Code

```
// Map Input Ports
const real_T *stage_in  = ssGetInputPortSignal(S,0);
const real_T *velocity  = ssGetInputPortSignal(S,1);
const real_T *accel     = ssGetInputPortSignal(S,2);
const real_T *ignflg    = ssGetInputPortSignal(S,3);
const real_T *boost_in  = ssGetInputPortSignal(S,4);

// Map Output Ports
real_T *stage     = ssGetOutputPortSignal(S,0);
real_T *resetflg  = ssGetOutputPortSignal(S,1);
real_T *boost     = ssGetOutputPortSignal(S,2);

// Initialize local variables
double vmag     = 0;
double vthresh  = 15;
double amag     = 0;
double athresh  = 5;

// Preliminary Calculations
vmag    = sqrt(pow(velocity[0],2)+pow(velocity[1],2)+pow(velocity[2],2));
amag    = sqrt(pow(accel[3],2)+pow(accel[4],2)+pow(accel[5],2));

*stage  = *stage_in;

if (*ignflg == 11)
{
        if (vmag < vthresh)
        {
                *resetflg = 1;
                *stage    = 1;
        }
}
if (*ignflg == 10)
{
        *resetflg = 0;
}

if (*ignflg == 21)
{
        if (amag < athresh)
        {
                *resetflg = 1;
                *stage    = 2;
        }
}
if (*ignflg == 20)
{
        *resetflg = 0;
}

if (*ignflg == 31)
{
        if (amag < athresh)
        {
                *resetflg = 1;
                *stage    = 3;
        }
}
if (*ignflg == 30)
{
        *resetflg = 0;
}

if (*ignflg == 41)
{
        if ( amag < athresh)
```

```
        {
                *resetflg = 1;
                *stage    = 4;
        }
}
if (*ignflg == 40)
{
        *resetflg = 0;
}

boost[0] = boost_in[0];
boost[1] = boost_in[1];
boost[2] = boost_in[2];
```

A.7 Interlocks

The Interlocks function simply determines which parts of the missile model to activate based upon the staging commands accepted from Vehicle Control. By setting the appropriate flags, the Interlocks function "ejects" the empty stages from the missile model as they are exhausted and begins consumption of the next remaining stage.

A.7.1 Interlocks Code

```
// Map Input Ports
const real_T *stage     = ssGetInputPortSignal(S,0);

// Map Output Ports
real_T *interlocks      = ssGetOutputPortSignal(S,0);

// Initialize local variables

// Staging Logic
if (*stage == 1)
{
        // Command Booster Ignition
        interlocks[0]   = 1;  // 1st Stage Motor Interlock (Ignite)
        interlocks[1]   = 0;  // 1st Stage Ejection Ordnance Interlock
        interlocks[2]   = 0;  // 2nd Stage Motor Interlock
        interlocks[3]   = 0;  // 2nd Stage Ejection Ordnance Interlock
        interlocks[4]   = 0;  // 3rd Stage Motor Interlock
        interlocks[5]   = 0;  // 3rd Stage Ejection Ordnance Interlock
}
else if (*stage == 2)
{
        // Transition to Stage 2
        interlocks[0]   = 0;
        interlocks[1]   = 1;  // Eject Stage 1
        interlocks[2]   = 1;  // Ignite Stage 2 Motor
        interlocks[3]   = 0;
        interlocks[4]   = 0;
        interlocks[5]   = 0;
}
else if (*stage == 3)
{
        // Transition to Stage 3
        interlocks[0]   = 0;
        interlocks[1]   = 1;
        interlocks[2]   = 0;
        interlocks[3]   = 1;  // Eject Stage 2
        interlocks[4]   = 1;  // Ignite Stage 3 Motor
        interlocks[5]   = 0;
}
else if (*stage == 4)
{
        // Transition to Post-Boost
        interlocks[0]   = 0;
        interlocks[1]   = 1;
        interlocks[2]   = 0;
        interlocks[3]   = 1;
        interlocks[4]   = 0;
        interlocks[5]   = 1;  // Eject Stage 3
}
```

A.8 Velocity Measurement

In this simplified simulation, the Velocity Measurement function performs the basic function of an ideal accelerometer. It samples the missile model's output of specific force, and accumulates measurements of the missile's change in velocity at 100-Hz (execution rate hardcoded into the simulation). This function passes the measurements and recorded time interval to Navigation in the form of a 4-element vector (in the inertial frame).

A.8.1 Velocity Measurement Code

```
// Map Input Ports
const real_T *f2        = ssGetInputPortSignal(S,0);
const real_T *f1        = ssGetInputPortSignal(S,1);
const real_T *time      = ssGetInputPortSignal(S,2);
const real_T *dvsum     = ssGetInputPortSignal(S,3);

// Map Output Ports
real_T *delta_v = ssGetOutputPortSignal(S,0);
real_T *f_last  = ssGetOutputPortSignal(S,1);

// Initialize Internal Variables
double dv[3]    = {0};
double dt       = 0;

// Calculate Outputs
dt = *time - dvsum[0];

// Measure Change-in-Velocity Since Last Measurement
dv[0]   = (f2[0]+f1[0])*dt/2;
dv[1]   = (f2[1]+f1[1])*dt/2;
dv[2]   = (f2[2]+f1[2])*dt/2;

// Add to Accumulated Sum
delta_v[0]   = *time;
delta_v[1]   = dv[0] + dvsum[1];
delta_v[2]   = dv[1] + dvsum[2];
delta_v[3]   = dv[2] + dvsum[3];

// Store Measurements from this Pass
f_last[0]    = f2[0];
f_last[1]    = f2[1];
f_last[2]    = f2[2];
```

Appendix B

Candidate System Missile Model

B.1 Missile Design Script

The missile design script generates sample propellant and stage structure weights, as well as propellant flow rates and stage burn times, according to parameters as specified by the user. User inputs include propellant specific impulse, stage mass-fuel fractions, maximum axial accelerations, and a desired final velocity (actually a change in velocity). The data generated by this script was used to construct the three-stage missile model that follows. This algorithm was extracted from pp. 265-279 of Zarchan [8].

```
%% Thrust-Weight Profile Script

%% Load Constants
g       = 32.2;

TWP.isp1    = 280;
TWP.isp2    = 280;
TWP.isp3    = 280;
TWP.mf1     = 0.85;
TWP.mf2     = 0.85;
TWP.mf3     = 0.85;
TWP.amax1   = 15;
TWP.amax2   = 15;
TWP.amax3   = 15;

TWP.wpd     = 600*8;

TWP.delta_v = 22000;
TWP.dv1     = (1/3)*(TWP.delta_v);
TWP.dv2     = (1/2)*(TWP.delta_v);
TWP.dv3     = (1/6)*(TWP.delta_v);

%% Calculations
```

```
%% Third Stage
TWP.top     = TWP.wpd*(exp(TWP.dv3/(g*TWP.isp3)) - 1);
TWP.bot     = 1/TWP.mf3 - ((1-TWP.mf3)/TWP.mf3)*exp(TWP.dv3/(g*TWP.isp3));
TWP.wp3     = TWP.top/TWP.bot;
TWP.ws3     = TWP.wp3*(1-TWP.mf3)/TWP.mf3;
TWP.wt3     = TWP.wp3 + TWP.ws3 + TWP.wpd;

%% Second Stage
TWP.top     = TWP.wt3*(exp(TWP.dv2/(g*TWP.isp2)) - 1);
TWP.bot     = 1/TWP.mf2 - ((1-TWP.mf2)/TWP.mf2)*exp(TWP.dv2/(g*TWP.isp2));
TWP.wp2     = TWP.top/TWP.bot;
TWP.ws2     = TWP.wp2*(1-TWP.mf2)/TWP.mf2;
TWP.wt2     = TWP.wp2 + TWP.ws2 + TWP.wt3;

%% First Stage
TWP.top     = TWP.wt2*(exp(TWP.dv1/(g*TWP.isp1)) - 1);
TWP.bot     = 1/TWP.mf1 - ((1-TWP.mf1)/TWP.mf1)*exp(TWP.dv1/(g*TWP.isp1));
TWP.wp1     = TWP.top/TWP.bot;
TWP.ws1     = TWP.wp1*(1-TWP.mf1)/TWP.mf1;
TWP.wt      = TWP.wp1 + TWP.ws1 + TWP.wt2;

%% Thrust Levels
TWP.T1 = TWP.amax1*(TWP.wt2 + TWP.ws1);
TWP.T2 = TWP.amax2*(TWP.wt3 + TWP.ws2);
TWP.T3 = TWP.amax3*(TWP.wpd + TWP.ws3);

%% Burn Times
TWP.bt1 = TWP.isp1*TWP.wp1/TWP.T1;
TWP.bt2 = TWP.isp2*TWP.wp2/TWP.T2;
TWP.bt3 = TWP.isp3*TWP.wp3/TWP.T3;

%% Flow Rates
TWP.fr1 = TWP.wp1/TWP.bt1;
TWP.fr2 = TWP.wp2/TWP.bt2;
TWP.fr3 = TWP.wp3/TWP.bt3;

disp(sprintf('=============================================================================='));
disp(sprintf('Missile Specifications:'));
disp(sprintf('=============================================================================='));
disp(sprintf('Based on ->         \t ISP \t\t\t MF \t\t\tAMAX'));
disp(sprintf('            STAGE 1:\t %0.0f \t\t\t%0.2f\t\t\t %0.0f ', TWP.isp1, TWP.mf1, TWP.amax1));
disp(sprintf('            STAGE 2:\t %0.0f \t\t\t%0.2f\t\t\t %0.0f ', TWP.isp2, TWP.mf2, TWP.amax2));
disp(sprintf('            STAGE 3:\t %0.0f \t\t\t%0.2f\t\t\t %0.0f ', TWP.isp3, TWP.mf3, TWP.amax3));
disp(sprintf('=============================================================================='));
disp(sprintf('                                        '));
disp(sprintf('\t         \tPAYLOAD                    '));
disp(sprintf('\t         \t=======                    '));
disp(sprintf('\t  _____  \t  WEIGHT:\t %0.2f', TWP.wpd   ));
disp(sprintf('\t /XXXXX\\                               '));
disp(sprintf('\t|       |\tSTAGE 3                    '));
disp(sprintf('\t|       |\t=======                    '));
disp(sprintf('\t|       |\t  PROP:\t\t %0.2f', TWP.wp3   ));
disp(sprintf('\t|XXXXXXX|\t  STRUC:\t  %0.2f', TWP.ws3   ));
disp(sprintf('\t|XXXXXXX|\t  BURN:\t\t    %0.2f', TWP.bt3));
disp(sprintf('\t|][][][]|\t  FLOW:\t\t  %0.2f', TWP.fr3  ));
disp(sprintf('\t|XXXXXXX|                               '));
disp(sprintf('\t|XXXXXXX|\tSTAGE 2                    '));
disp(sprintf('\t|XXXXXXX|\t=======                    '));
disp(sprintf('\t|       |\t  PROP:\t\t%0.2f', TWP.wp2    ));
disp(sprintf('\t|       |\t  STRUC:\t %0.2f', TWP.ws2    ));
disp(sprintf('\t|XXXXXXX|\t  BURN:\t\t   %0.2f', TWP.bt2 ));
disp(sprintf('\t|XXXXXXX|\t  FLOW:\t\t  %0.2f', TWP.fr2  ));
disp(sprintf('\t|XXXXXXX|                               '));
disp(sprintf('\t|XXXXXXX|\tSTAGE 1                    '));
disp(sprintf('\t|XXXXXXX|\t=======                    '));
disp(sprintf('\t|XXXXXXX|\t  PROP:\t\t%0.2f', TWP.wp1    ));
disp(sprintf('\t|XXXXXXX|\t  STRUC:\t%0.2f', TWP.ws1     ));
disp(sprintf('\t|XXXXXXX|\t  BURN:\t\t   %0.2f', TWP.bt1 ));
```

```
disp(sprintf('\t|XXXXXXX|\t  FLOW:\t\t %0.2f', TWP.fr1   ));
disp(sprintf('\t|XXXXXXX|                              '));
disp(sprintf('\t  /___\\                              '));
disp(sprintf('\t         \tTOTAL                      '));
disp(sprintf('\t         \t=======                    '));
disp(sprintf('\t         \t       \t   %0.2f', TWP.wt    ));
disp(sprintf('================================================================================'));
```

B.2 Missile Script Output

```
================================================================================
Missile Specifications:
================================================================================
Based on ->              ISP             MF              AMAX
           STAGE 1:      280           0.85              15
           STAGE 2:      280           0.85              15
           STAGE 3:      280           0.85              15
================================================================================

                     PAYLOAD
                     =======
        _____          WEIGHT:        4800.00
       /XXXXX\
      |       |      STAGE 3
      |       |      =======
      |       |        PROP:          2642.83
      |XXXXXXX|        STRUC:          466.38
      |XXXXXXX|        BURN:             9.37
      |][][][]|        FLOW:           282.13
      |XXXXXXX|
      |XXXXXXX|      STAGE 2
      |XXXXXXX|      =======
      |       |        PROP:         32628.66
      |       |        STRUC:         5758.00
      |XXXXXXX|        BURN:            44.56
      |XXXXXXX|        FLOW:           732.17
      |XXXXXXX|
      |XXXXXXX|      STAGE 1
      |XXXXXXX|      =======
      |XXXXXXX|        PROP:         74667.27
      |XXXXXXX|        STRUC:        13176.58
      |XXXXXXX|        BURN:            23.44
      |XXXXXXX|        FLOW:          3186.02
      |XXXXXXX|
       /___\
                     TOTAL
                     =======
                              134139.71
================================================================================
```

B.3 Missile Block Diagram

The missile block diagram model (constructed in Simulink) is decomposed in the following pages. The missile model consists of an interlocks model that activates the appropriate stages and motors, three stage motor models, and a 3DOF equations of motion model. Inputs to the missile model are staging commands and a boost vector from Vehicle Control, and the outputs are missile true acceleration, velocity, and position, as well as specific force for measurement by the Velocity Measurement function. The interlocks model is constructed in ANSI C, and can be viewed in Appendix Section A.7.

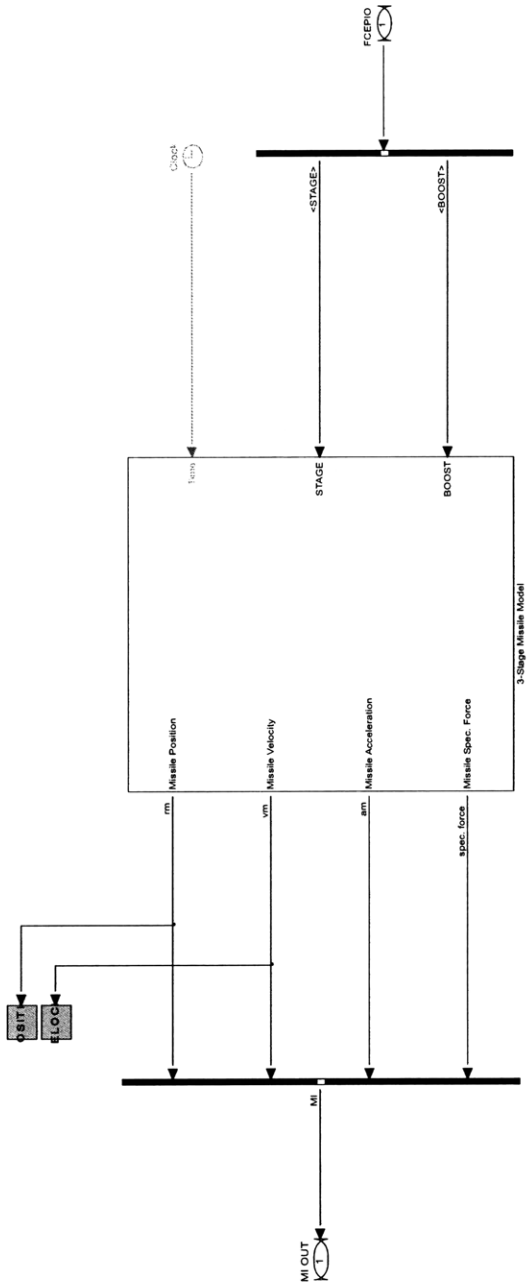

Figure B-1: Missile Model - Top Level

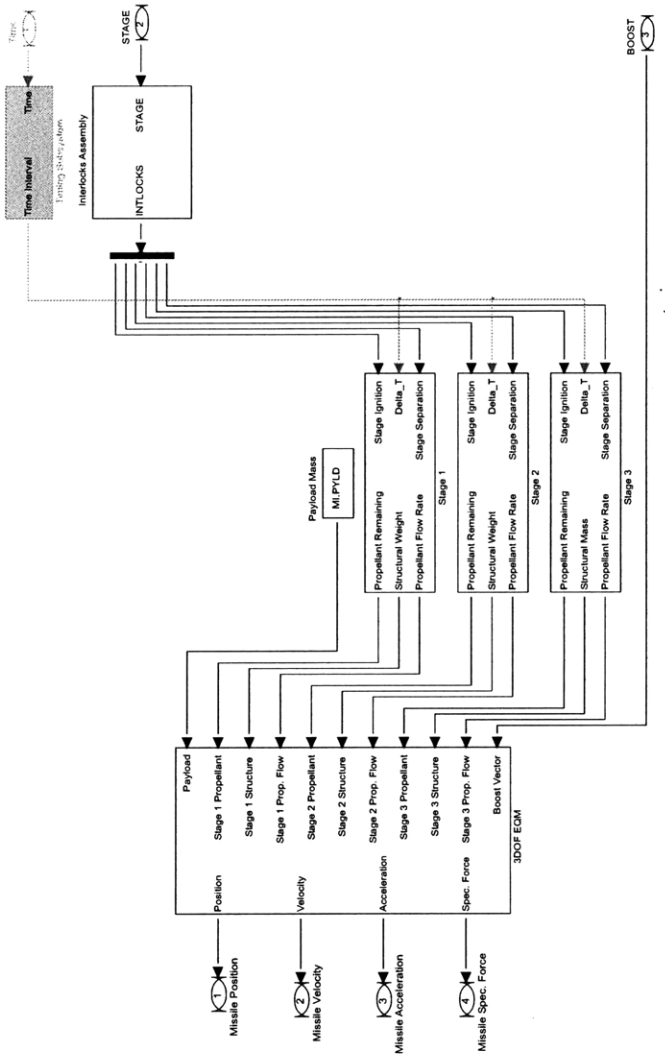

Figure B-2: Missile Model - Interlocks, Staging, and 3DOF Dynamics

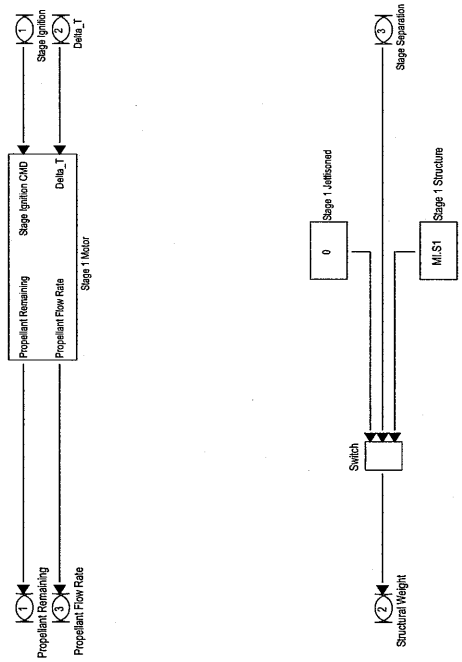

Figure B-3: Missile Model - Sample Stage (Stage 1)

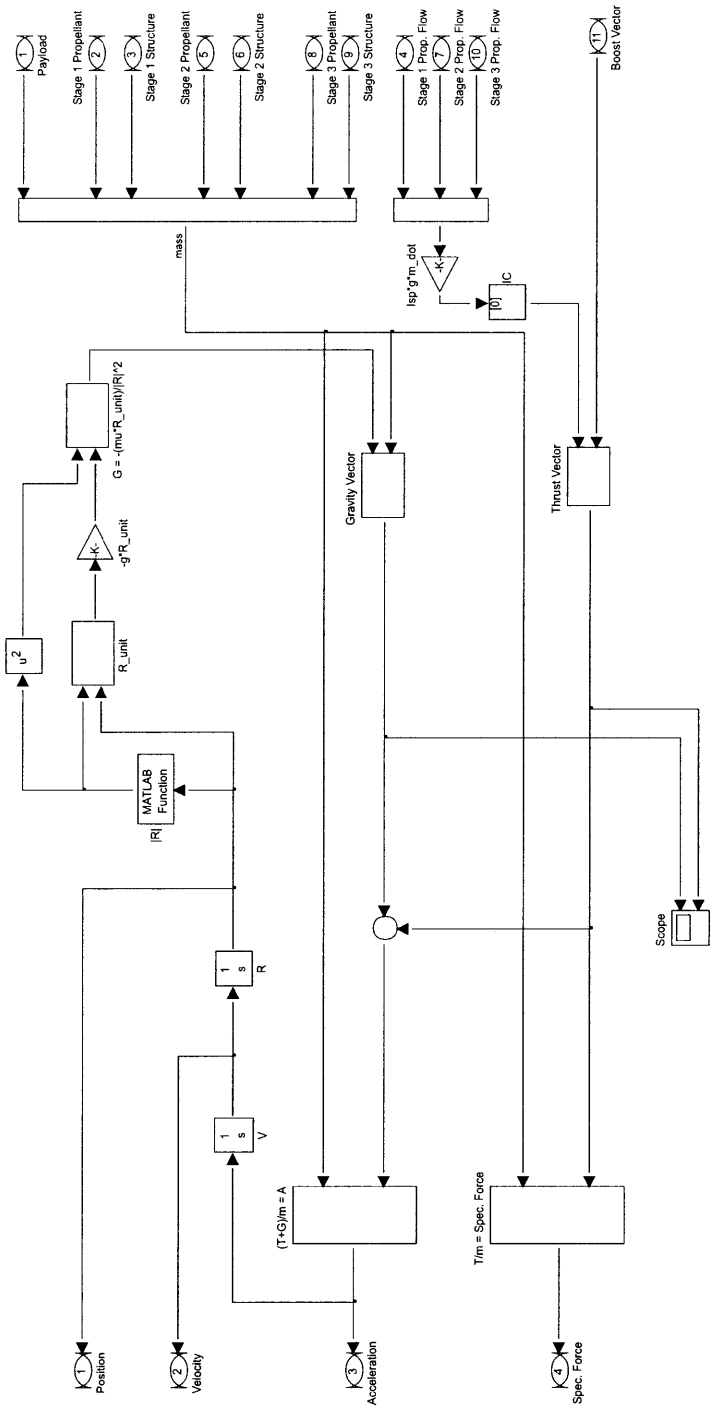

Figure B-4: Missile Model - 3DOF Equations of Motion

[This page intentionally left blank.]

Appendix C

Simulation Hierarchical Decomposition: Vertical Architecture

This appendix contains screenshots of the hierarchical decomposition of the vertical simulation architecture in Simulink. The functionally-oriented organization of this architecture is clearly shown in Figure C-3. Figure C-4 demonstrates the concept of grouping all functions of a particular class in the same subsystem. The "function selection logic" uses the current mode to determine which function to execute, and the data is collected and passed as a single subsystem output by the blue "merge" block. The MSC function has been used to demonstrate these concepts; all other function classes are identical in form. The rest of the figures depict each function and its interfaces as implemented in the simulation.

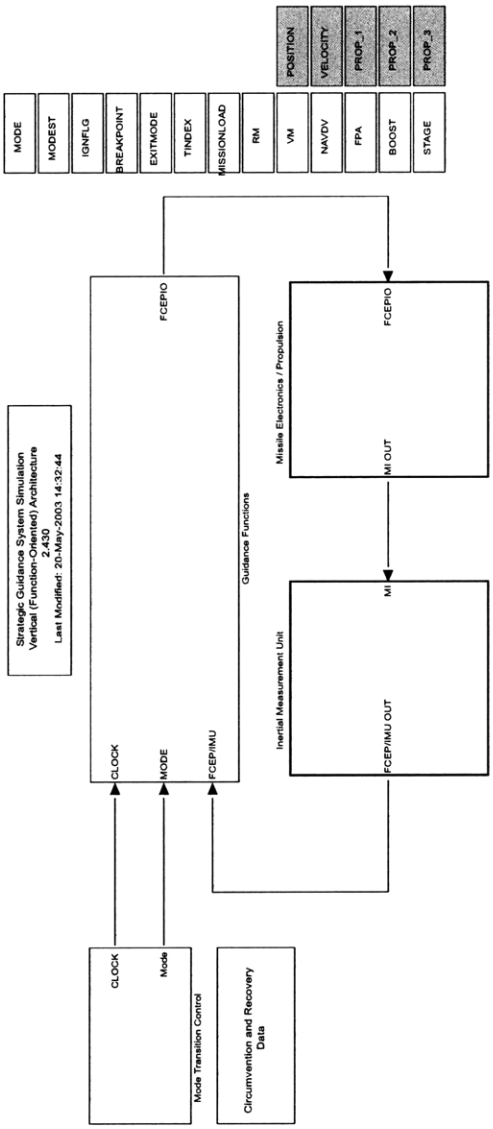

Figure C-1: Vertical Simulation Architecture - Top Level

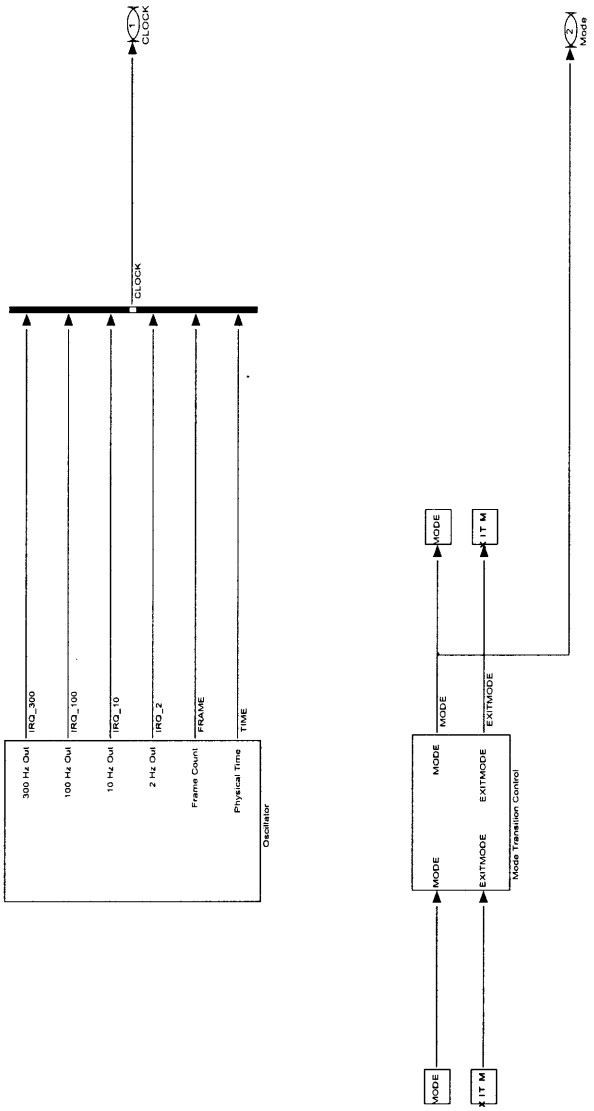

Figure C-2: Vertical Simulation Architecture - Mode Transition Control & Oscillator

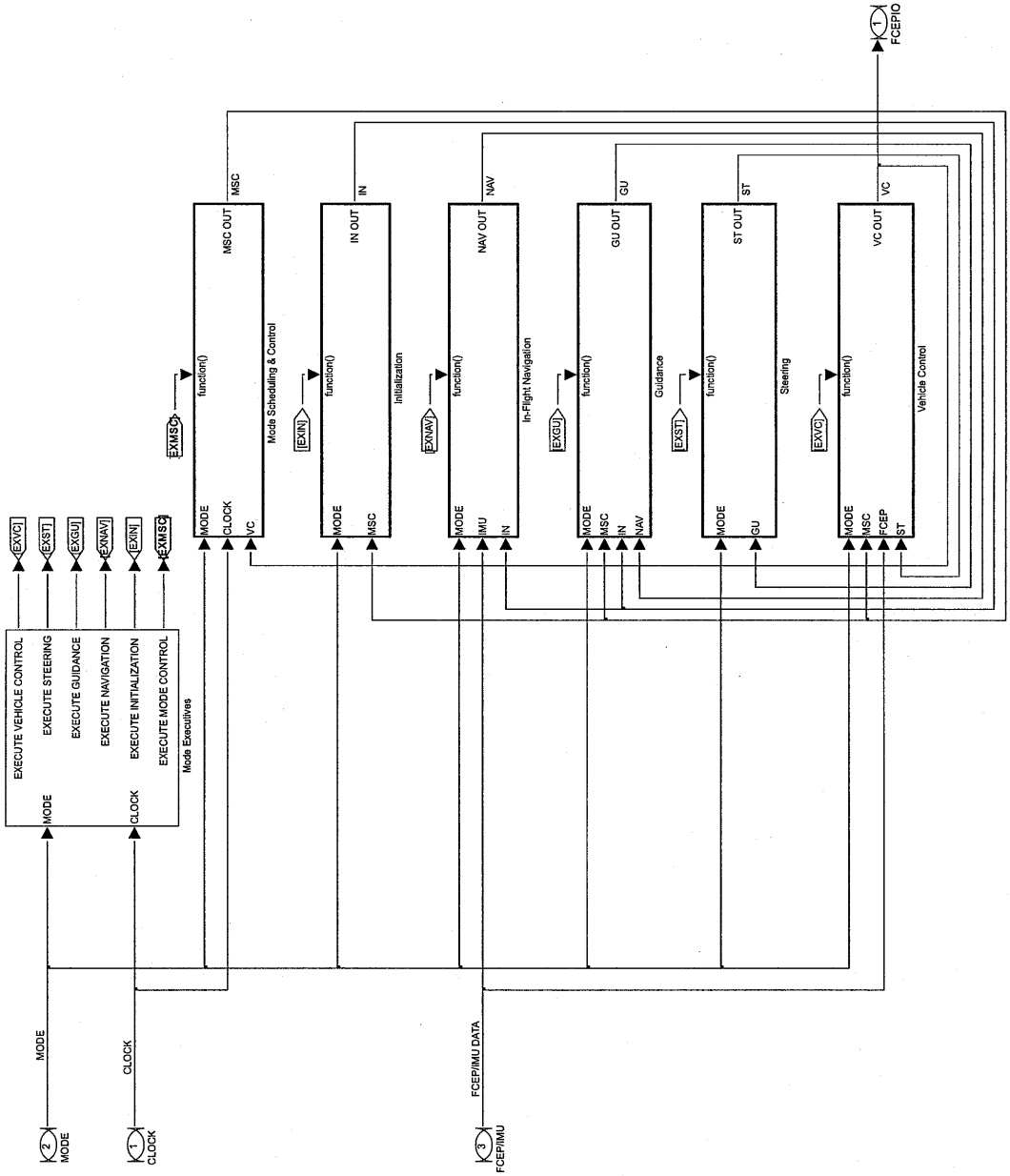

Figure C-3: Vertical Simulation Architecture - Functions Top Level

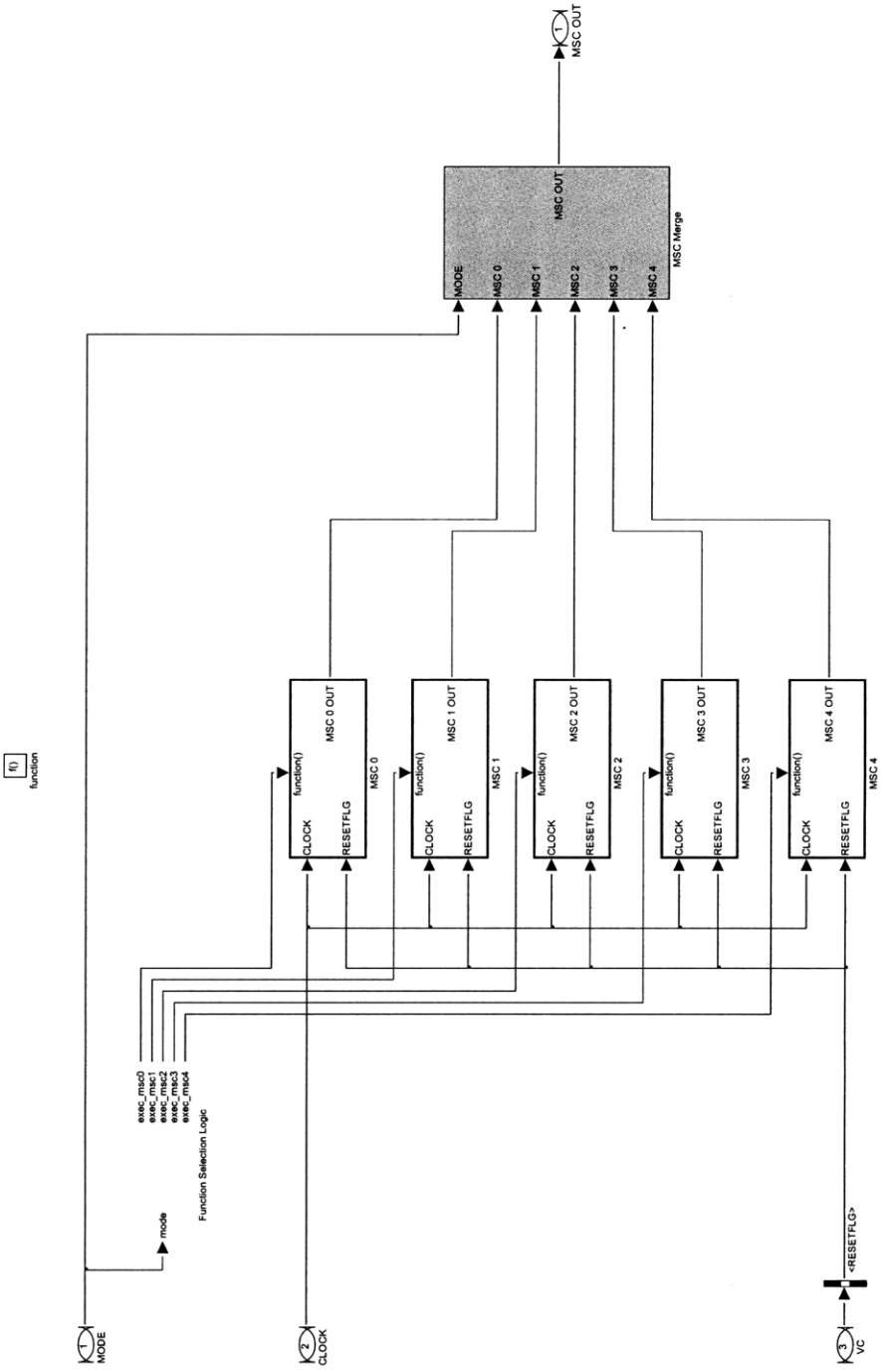

Figure C-4: Vertical Simulation Architecture - MSC Function Class

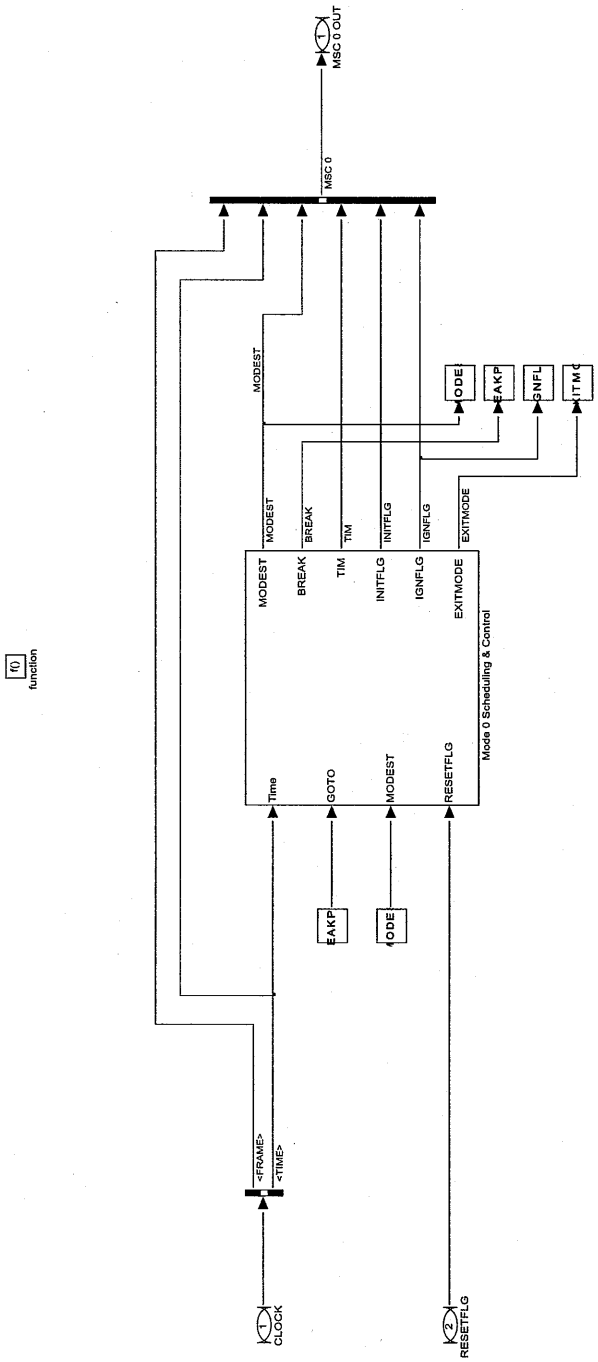

Figure C-5: Vertical Simulation Architecture - MSC Function (Mode 0)

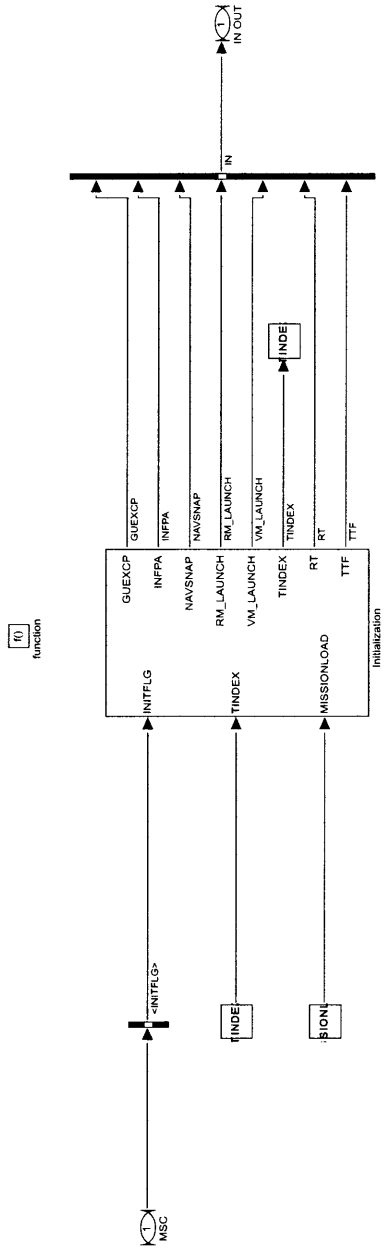

Figure C-6: Vertical Simulation Architecture - Initialization

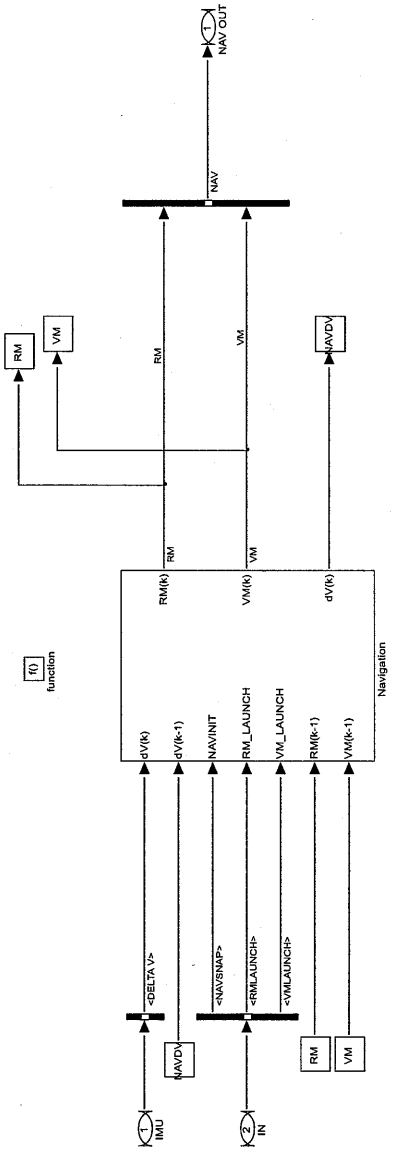

Figure C-7: Vertical Simulation Architecture - Navigation

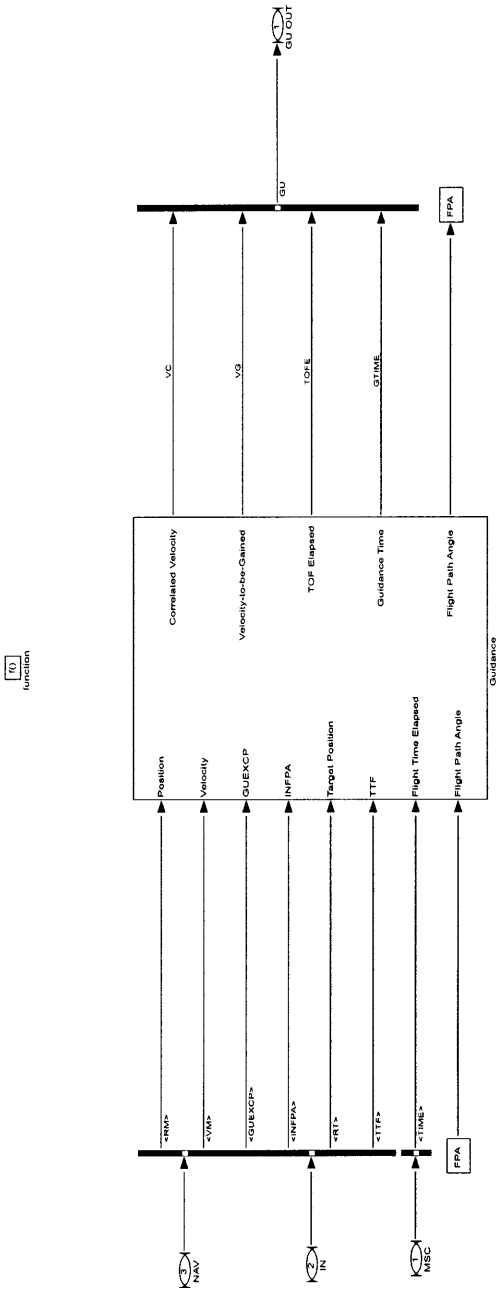

Figure C-8: Vertical Simulation Architecture - Guidance

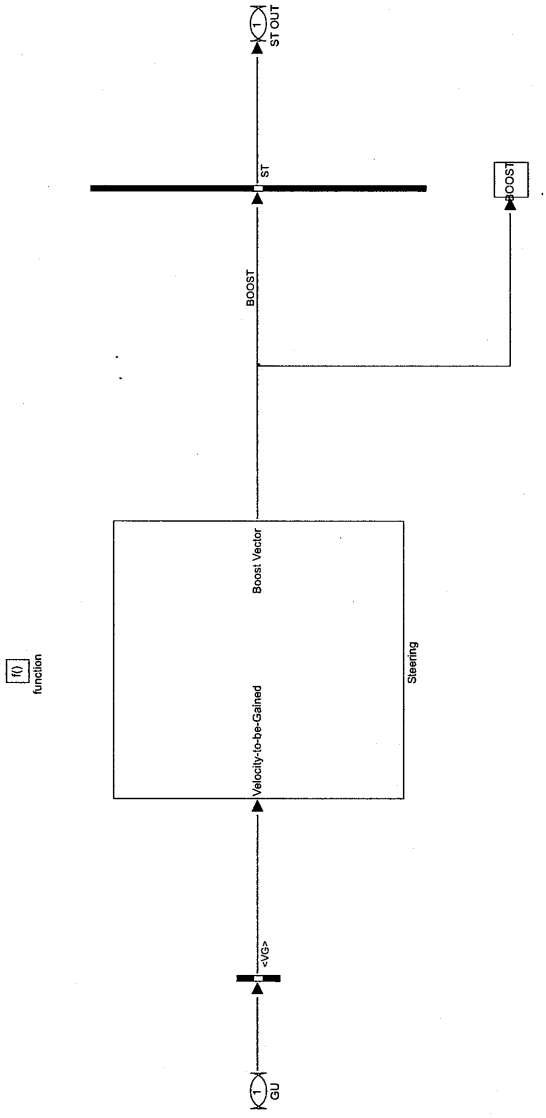

Figure C-9: Vertical Simulation Architecture - Steering

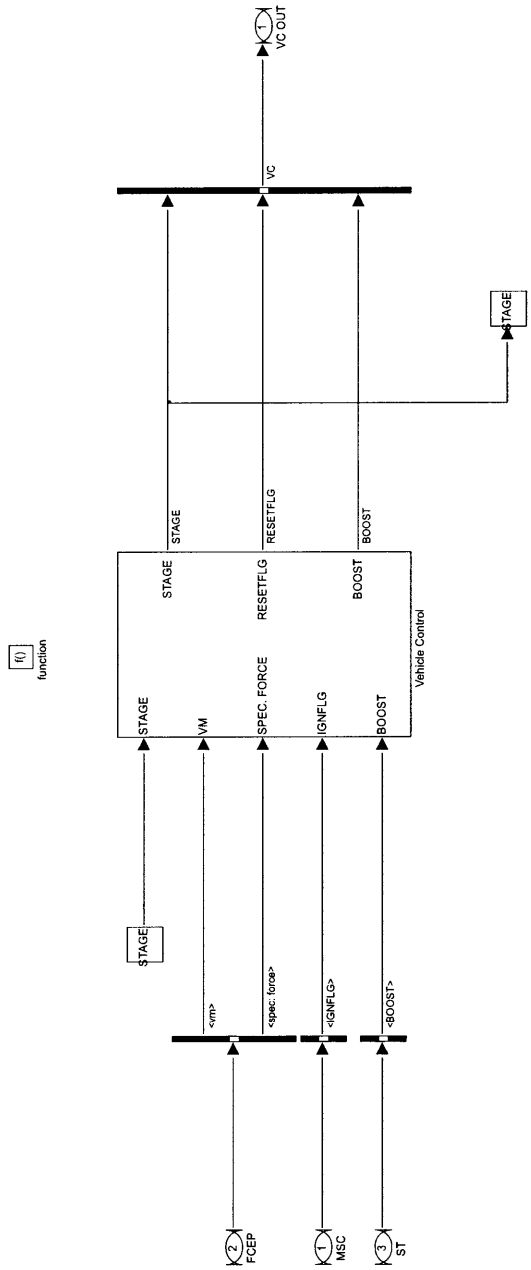

Figure C-10: Vertical Simulation Architecture - Vehicle Control

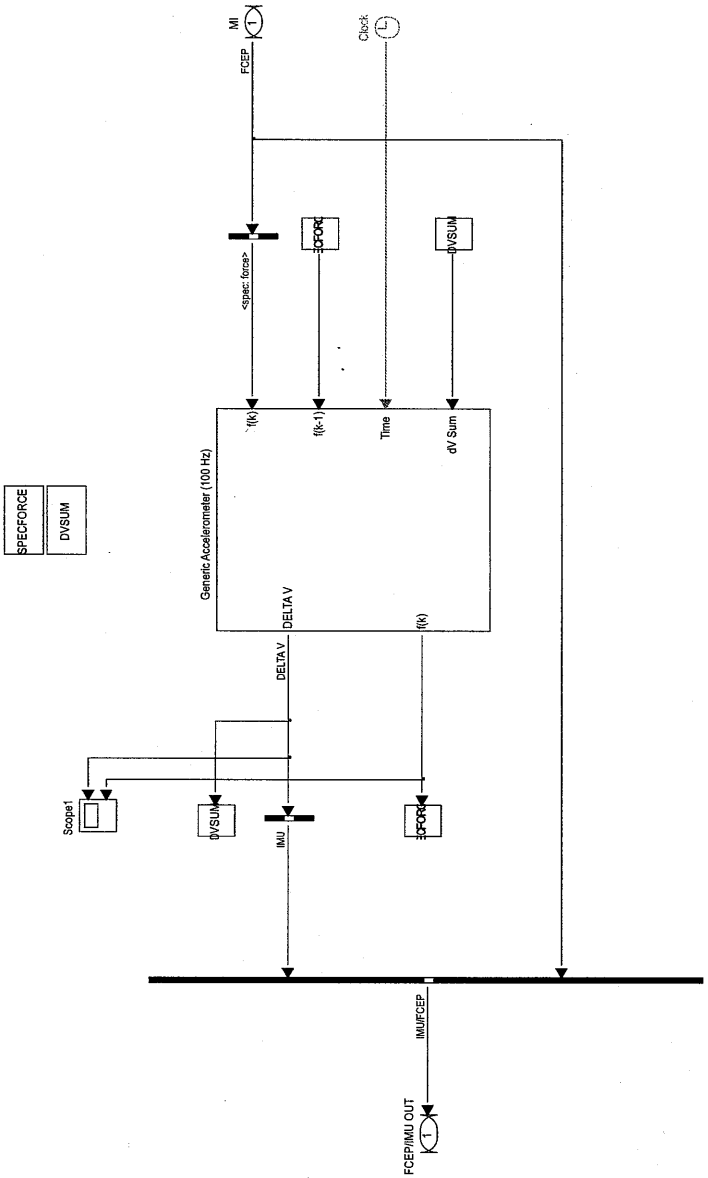

Figure C-11: Vertical Simulation Architecture - Velocity Measurement

References

[1] DoD Directive 5000.59-M. *Defense Modeling and Simulation (M&S) Glossary.* U.S. Department of Defense, January 1998.

[2] DoD Directive 5000.59-P. *Defense Modeling and Simulation (M&S) Master Plan.* U.S. Department of Defense, October 1995.

[3] Bob Aronson and Mike Woods. Trident hull and missile life extensions approved. *Undersea Warfare*, 1(1), Fall 1998.

[4] Averil B. Chatfield. *Fundamentals of High Accuracy Inertial Navigation*, volume 174 of *Progress in Astronautics and Aeronautics.* American Institute of Aeronautics and Astronautics, Inc., Reston, Virgina, 1997.

[5] Michael V.R. Johnson, Sr., Mark F. McKeon, and Terence R. Szanto. Simulation-based acquisition: A new approach. Technical report, Defense Systems Management College, FT Belvoir, Virginia, December 1998.

[6] The MathWorks, Inc. *Using Simulink: Version 5*, July 2002.

[7] Graham Spinardi. *From Polaris to Trident: The development of US Fleet Ballistic Missile technology.* Cambridge University Press, 1994.

[8] Paul Zarchan. *Tactical and Strategic Missile Guidance*, volume 157 of *Progress in Astronautics and Aeronautics.* American Institute of Aeronautics and Astronautics, Inc., Reston, Virginia, second edition, 1994.